Git

گردآورنده:
Payam Avarwand

مقدمه:

درحقیقت کارکرد Git، مشابه به یک سیستم Backup ریزبینانه است که می توان از آن جهت پشتیبان گیری و کنترل تغییرات هرگونه فایلی (بطورخاص، فایل های متنی) استفاده کرد.

اما از آنجایی که فلسفه ی ایجاد Git، یک سیستم پشتیبان گیری برای طولانی مدت و کنترل تغییرات به شکل کاملا جز به جز و دقیق، طوری که بتوان کدهای برنامه نویس ها را مدیریت و تغییرات آنها را ذخیره و حفظ کرد، بوده است، کاربرد اصلیGit ، در زمینه ی پشتیبانی از Script ها و Code های کامپیوتری است.
اما لازم است که بدانیم، هرکسی می تواند از Git استفاده کند، مخصوصن نویسنده ها!

همانطور که احتمالاً مشاهده می کنید، این کتابچه دارای دسته بندی های منظم و سلسله مراتبی است که پس از باز کردن آن مثلاً توسط نرم افزار Adobe Reader ، و با باز کردن تب Bookmarks، به سادگی به سرتیترهای این دسته بندی دسترسی خواهید داشت که باعث سرعت عمل در کار با کتاب و دسترسی به موضوعات مختلف آن، خواهد بود.

Practical Glossary:

Phrase / Abbreviation	Meaning	Description
Amend	اصلاح کردن	
Append	افزودن، ضمیمه کردن	
Appropriate	تخصیص دادن / متناسب، در خور	
Appropriation	تخصیص	
Assign	معین کردن، اختصاص دادن، واگذار کردن	
audit	بازرسی و بررسی کردن	
Availability	در دسترس بودن	
Barebone System	سیستم نیمه آماده	کامپیوتری که فقط شامل قطعات اصلی راه اندازی است.
Built-In	از پیش ساخته شده	
Chip	تراشه	
Credential	مشخصات یک حساب کاربری	به مشخصات یک User که شامل نام و رمزعبور است، گفته می شود.
delegate	محول کردن، وکالت دادن	
Determine	مشخص کردن، معلوم کردن	
DSRM	Directory Services Restore Mode	حالتی که ویندوز Server از Domain جداست. مثل safe Mode
hybrid routing	مسیریابی دو طرفه، ارتباط دوطرفه	
in-house Server	سرور خانگی	
interactive	تعاملی (به شکلی که گفتگو/مذاکره شود.)	
Invoke	فراخواندن، فراخوانی کردن	
Jbod	Just a Bunch of Disks	NRaid or Spanned Raid
Junk Data	داده ی زائد	
On premise	Local، در محل	
Platform	بستر، محیط	
Prerequisite	پیش نیاز	
Quota	سهم	
Redundancy	زاپاس داشتن، ذخیره و جایگزین داشتن	
Repository	مخزن	
Statement of work : SOW	شرح کار	
Symmetric	متقارن	
Terminology	اصطلاح شناسی فنی | واژه شناسی	
text cursor	نشانگر متن	
to deploy	آماده به کار کردن	
UNC Address	Universal Naming Convention	\\DCName\FolderName
To **power cycle** a device	Turn off and unplug the device and wait at least 30 sec. and plug it in and turn it back on	

1 Bit	0 | 1	
1 Byte	8 Bits	
128 Bytes	1 Kilo Bit	1024 Bits
1 Kilo Byte	1024 Bytes	8 Kilo Bits
1 Mega Byte	1024 Kilo Bytes	1,048,576 Bytes
1 Giga Byte	1024 Mega Bytes	1,073,741,824 Bytes
1 Tera Byte	1024 Giga Bytes	1,048,576 Mega Bytes
1 Peta Byte	1024 Tera Bytes	1,048,576 Giga Bytes
1 Exa Byte	1024 Peta Bytes	1,048,576 Tera Bytes

: Practical Signs

Sign	Name
/	Slash
\	Backslash
|	Pipe
,	comma
:	Colon
;	semicolon
" "	double quotes (quotation marks)
‘ ‘	single quotes
'	Apostrophe
<	left angle bracket
>	right angle bracket
=	equals sign
&	Ampersand
≈	almost equal to
*	Asterisk
@	At Sign
.	Bullet
^	Caret
√	Checkmark
{	Curly Brackets
}	Left Curly Bracket
°	Degree
¨	Umlaut
÷	Division Sign
-	EN Dash
_	EM Dash
!	Exclamation Mark
♀	Female Sign
½	One half fraction
¼	One Quarter fraction
⅓	One third fraction
¾	Three quarters fraction
>	Greater Than Sign
∞	Infinity
<	Less Than Sign
♂	Male Sign
×	Multiplication Sign
#	Number Sign
Ω	Ohm Sign
(	Parenthese
)	Left Parenthesis
%	Percent Sign
£	Pound Sign
?	Question Mark
"	Quotation Marks
§	Section Sign
[	Square Brackets
]	Left Bracket

Introduction:

Git:

می دانیم که وقتی یک برنامه نوشته می شود، در طول زمان، برحسب نیازمندی ها و جهت رفع مشکلات و خطاها، مدام تغییراتی روی آن داده شده و منتشر می شود. هربار که تغییراتی روی آن برنامه ایجاد و سپس دوباره منتشر یا به روزرسانی می شود، یک نسخه (Version) محسوب می شود.
از طرف دیگر، این یک ضرورت است که نسخه های قبلی از دست نروند، و قابلیت بازگشت یا بررسی نسخه های قبلی وجود داشته باشد.

- **کاربرد اصلی Git، وجود محیطی برای کنترل Version و همکاری تیمی روی پروژه هاست.**
- **درحقیقت Git، یک سیستم Version Control است.**
- **سیستم های Version Control دیگری هم وجود دارند، مثل: Mercurial و Subversion**
- **در گیت، هربار که پس از اعمال تغییرات، پروژه را دوباره ذخیره می کنیم، درحقیقت، تمام پروژه ذخیره نمی شود! بلکه تنها تغییراتی که از آخرین نسخه ی ذخیره شده تا به حال اعمال کرده ایم، ذخیره می شوند، و در صورت نیاز به بازگشت به یکی از نسخه های ذخیره شده، فایل کامل پروژه، همراه با تغییراتی که در نسخه ی موردنظر ذخیره کرده ایم، بالا می آیند و به سادگی، به نسخه ی موردنظر باز می گردیم.**

- **Version Control در گیت‌هاب توسط Git انجام می‌شود.**
- **واژه‌ی Hub، به مفاهیم شبکه اشاره دارد که نشانه ی همکاری تیمی و ایجاد انشعاب در پروژه‌ها می باشد.**

اگر پروژه‌ای Open Source باشد، توسعه دهندگان دیگر می‌توانند آن را دانلود کرده، تغییراتی روی آن اعمال کنند و سپس پروژه را با ورژن جدید آپلود نمایند.

Git Terminology :

- Repository: **که به اختصار Repo هم نامیده می شود، به معنای مخزن است.**
  پروژه ها، درون Repo ذخیره می شوند.
  به عبارتی، به مکان ذخیره ی هر پروژه، Repository گفته می شود.
  در این مخزن می‌توان هر فولدر یا فایلی را با فرمت دلخواه ایجاد کرد.
  پس Repo فضایی است که در آن هر چیزی که مربوط به یک پروژه است، گردآوری و سازمان دهی می‌شود.
  - **Object هایی مثل Picture، Excel sheet ها، Video ها و هر چیزی که برای راه اندازی یک پروژه به آن نیاز است.**

- Branch یا Fork: **به معنای شاخه یا انشعاب است.**
  دو قابلیت مهمی که Branch ها دارند:
  - با این قابلیت می توان بطور اشتراکی (چند نفره) روی یک پروژه کار کرد.
  - می‌توان از یک پروژه ی موجود، یک انشعاب دریافت و تغییراتی را روی آن اعمال کرد و آن را به عنوان یک پروژه ی جدید منتشر کرد.

- Master: **به Fork اصلی در هر Repository، گفته می شود.**
  درحقیقت، Master همان default Branch مربوط به یک Repo می باشد.
  Master می تواند چندین زیرشاخه داشته باشد؛ که می توانیم به دلخواه روی یکی از این زیرشاخه ها کار کنیم.

- Pull Request: **که به معنای «درخواست ادغام» است، زمانی استفاده می‌شود که از پروژه‌ی اصلی، یک شاخه دریافت و درآن تغییراتی اعمال کرده باشیم.**
  به کمک Pull Request می‌توان به شخص ایجاد کننده‌ی پروژه ی اصلی، درخواست اعمال تغییرات را در پروژه ی اصلی داد.

- <u>Commit</u>: زمانی که تصمیم می گیریم تغییرات اعمال شده را به عنوان یک نقطه ی قابل بازیابی، ذخیره کنیم، به این تغییر ذخیره شده، یک commit گفته می شود. Commit در Git مشابه به SnapShot در مجازی سازی است. درحقیقت Repository، مجموعه ای از Commit هاست.

- <u>HEAD</u>: آخرین Commit انجام شده را با عنوان Head می شناسیم؛ و جهت اشاره به آخرین Commit، از Head استفاده می شود.
  - Head^ : جهت اشاره به Commit قبل از Head Commit استفاده می شود.
  - Head^^ : جهت اشاره به دو Commit قبل از Head Commit استفاده می شود.
  - Head~5 : جهت اشاره به پنج Commit قبل از Head Commit استفاده می شود.
  - . . .

- <u>Staging area</u>: **هرگاه که یک تغییر بر روی یک Data** (چه آن Data در Repo باشد، چه نباشد) **اعمال می کنیم، آن تغییر در Repo ثبت نمی شود، بلکه ابتدا در فضایی در خارج از Repo، به نام Staging area ثبت می شود، تا زمانی که آن را به Repo ارسال کنیم.**

- <u>Working tree</u>: **هم Directory یا مسیری است که پروژه** (Repo) **در آن قرار دارد و هم مجموعه Directory هایی که درون پوشه ی Repo تعریف کرده ایم.**

- <u>Origin</u>: **این اصطلاح را با UpStream را می شناسیم.**

درحقیقت به جایی که Repo را از آنجا دریافت کرده باشیم، Origin گفته می شود.

برای ما، دو حالت وجود دارد:

- یا می خواهیم به کمک Origin، Branch خودمان را update کنیم.
- و یا می خواهیم تغییراتی که خودمان اعمال کرده ایم را برای Origin ارسال کنیم.

بطورمثال، اگر پروژه ای را از یکی از سایت های Github یا Gitlab فراخوانی کرده باشیم، خود این سایت ها، Origin خواهند بود.

:Get started with Git

❖ **نصب و راه اندازی Git:**

- **در ابتدا (**در Windows**) بسته ی مربوط به این ابزار را دریافت و بر روی سیستم نصب می کنیم. پس وارد وبسایت** https://git-scm.com/ **شده و روی Downloads کلیک کرده و نسخه ی Windows را دانلود کرده و به سادگی نصب می کنیم.**
- **در پایان، اگر در cmd دستور** `git` **را وارد کنیم، به جای نمایش خطا، راهنمای دستور git را دریافت خواهیم کرد.**
- **در Linux، و در Terminal، توسط دستور** `git version` **می توان بررسی کرد که Git نصب است یا نه، و در صورت نیاز، توسط دستور زیر آنرا نصب می کنیم:**

Sudo apt-get install git

- **جهت بررسی نصب و فعال بودن GIT در CLI، هم می توان در cmd اقدام کرد و هم توسط محیط مخصوص خودش با ورود به آیکون میانبوری که پس از نصب، در Desktop اضافه شده است، به نام Git Bash.**
- **جهت کار با Git، هم می توان از Git Bash استفاده کرد، و هم از cmd.**

اما ابزار Git Bash کاربردی تر است، و اما اگر کسی هیچ آشنایی با دستورات Terminal در Linux ندارد، cmd ساده تر است.

وارد Git Bash می شویم.

نکته: تقریباً همه ی دستورهایی که در Terminal سیستم عامل Linux معتبر هستند، در Git bash هم اعتبار دارند.

نکته: دستوردهی در محیط Git دو مشخصه ی Case Sensitive و Auto Completion را نیز مانند محیط BASH پشتیبانی می کند.

نکته: وقتی Git Bash را باز می کنیم، درحقیقت Command Line به طورپیشفرض در مسیر پوشه ی حساب جاری است، یعنی: C:\Users*USERNNAME*

توصیه: به جای زبان Bash، می توان از زبان ZBash استفاده کرد، که در وبسایت https://ohmyz.sh/ **می توان مشخصات آن را دید.**

❖ **جهت نصب آن در سیستم عامل Linux:**

Sudo apt install zsh

جهت سوییچ به Z Shell کافی ست درج کنیم: zsh

❖ **جهت نصب آن در سیستم عامل Windows:**

1. **ابتدا نرم افزار رایگان فشرده سازی PeaZip را دانلود می کنیم.**
2. **نسخه ی ویندوز، فایل نصب Zshell را که دارای فرمت zst. می باشد، دانلود و توسط PeaZip باز و extract می کنیم.** (از اینجا)
3. **همه ی فایل های extract شده را در مسیر نصب Git** (C:\Program Files\Git) **کپی می کنیم.**
4. **Git Bash را باز کرده و درج می کنیم: zsh**

مزایای استفاده از Zshell:

- Plugin های خوب و مهم و مرتبط با Git
- قابلیت Command Completion جامع تر

پس از نصب Zsh، خط فرمان (CL) به شکل ~ تغییر پیدا می کند، که نشانه ی حضور در محیط Zsh می باشد.

✓ **برای بازگشت به محیط Bash، کافی ست یکبار دستور exit را وارد کنیم.**

✓ **برای بازگشت به محیط ZShell، کافی ست یکبار دستور zsh را وارد کنیم.**

اولین قدم در کار با Git، معرفی خودمان به Git است، چرا که ضروری است که Git بداند چه کسی در حال کار کردن با اوست، و چه کسی چه تغییری را اعمال کرده است.

❖ **پس می توانیم توسط دستور زیر، خودمان را معرفی کنیم:**

```
git config user.name "Payam"
```

اما پس از وارد کردن این دستور، سیستم Git، این نام کاربری ما را تنها در پروژه ی جاری ثبت خواهد کرد. پس برای اینکه یک Username را جهت معرفی در تمام محیط Git، تعیین کنیم، به شکل وارد می کنیم:

```
git config --global user.name "Payam"
```

❖ **حال لازم است که Email مان را نیز به سیستم معرفی کنیم:**

```
git config --global user.email "payam_avar@yahoo.com"
```

❖ جهت مشاهده ی تنظیمات وارد شده:

```
git Config --list
```

نکته: در حالت Global، تنظیمات وارد شده برای این Username در همه ی Repo ها صدق خواهد کرد؛ اما در حالت Local، تنها در Repo جاری خواهد بود.

نکته: حال اگر یک Username جدید ایجاد، کنیم، می بینیم که در تنظیمات، Username هم تغییر می کند.

نکته: اگر Config را کامل انجام نداده باشیم، نخواهیم توانست Commit انجام دهیم.

Terminal Help:

با توجه به اینکه Git Bash کاملن مشابه با Terminal موجود در سیستم عامل Linux است و ویرایشگرمتنی که استفاده می کند، همان VIM است، لازم است که با برخی از دکمه های میانبور در این ابزار آشنا باشیم.

VIM editor دو حالت دارد:

- **جهت ورود به حالت text mode** (حالتی که در آن می توان ارتباط مستقیم با محتویات متنی یک فایل برقرار کرد)، **لازم است که دکمه ی Insert را از روی کیبورد بفشاریم.**
- **جهت خروج از text mode، کافی ست دکمه ی Esc را بفشاریم، که به Command mode بازگردد.**

در command mode، توسط دکمه های زیر می توان متن را مدیریت کرد:

کارکرد	دکمه روی کیبورد
u	undo
j	moving the cursor down
k	moving the cursor up
h	moving the cursor to the left
l (L)	moving the cursor to the right right
H	Jumping directly on the top of the screen
M	Jumping in the middle of the screen
L	Jumping directly to the bottom of the screen
w	Jumping on the start of a written word
e	Jumping toward the end of a written word
:w	Save the changes
:wq	Exit and save the changes
:q!	Exit and discard the changes

❖ **جهت ورود به یک Directory و ایجاد یک Directory** ، مثلاً \:F:

```
Cd f:
```

نکته: ما می توانیم با دستور .. cd ، خط دستور را از Directory که در آن قرار دارد، یک قدم (پوشه) به عقب بازگردانیم.

❖ ایجاد یک فایل متنی:

```
touch Test1.txt
```

Repository:

توسط دستور mkdir یک پوشه ی دلخواه به نام Test1 را ایجاد می کنیم:

```
mkdir Test1
```

پوشه ی Test1 تنها یک پوشه ی معمولی است.

❖ **جهت معرفی یک پروژه** (که در یک پوشه ی معمولی قرار دارد)، **به Git، ابتدا وارد دایرکتوری آن می شویم، سپس:**

```
Git init
```

و در پاسخ، باید متنی مشابه زیر، ظاهر شود:

Initialized empty Git repository in F:/Test1/.git/

نکته: اگر Command Line، در آن Directory موردنظر نباشد، می توان با درج Directory آن، آن را به Git معرفی کرد:

```
Git init Test1/
```

درحقیقت، پس از معرفی این پوشه به عنوان یک پروژه به Git:

- ✓ آن پوشه ی معمولی، تبدیل به یک Git Repository خواهد شد.
- ✓ می بینیم که عبارت (master) در مقابل مسیر Directory درج می شود، که به معنای حضور در Branch اصلی آن Repo هستیم.
- ✓ پوشه ای مخفی به نام git. درون پوشه ی اصلی (Repo)، ایجاد می شود، که محل ذخیره ی فایل های کنترلی پروژه ی تحت Git از جمله اطلاعات Working Tree، خواهد بود، که نباید در آنها هیچگونه تغییری اعمال کرد.

نکته: اگر این پوشه ی git. را حذف کنیم، پوشه ی Test1 دوباره تبدیل به یک پوشه ی معمولی می شود!

❖ **مشاهده ی وضعیت:**

وقتی که Command Line، در هیچ Directory قرار ندارد، و یا در یک Directory غیر Repo قرار دارد، در صورت فراخواندن وضعیت، توسط دستور زیر، هیچ نتیجه ای دریافت نخواهیم کرد، اما وقتی که Command Line را وارد یک Repository کنیم:

❖ با وارد کردن دستور زیر، وضعیت سه مشخصه ی Branch و Commit و File های موجود را نشان می دهد:

```
Git status
```

نکته: بخش untracked files، فایل هایی را نشان می دهد که در پوشه ی Repo قرار دارند، اما در آن ثبت نشده اند؛ به این دلیل که یا به تازگی ایجاد شده اند، و یا فایل هایی هستند که از قبل بوده اند، اما تغییری در آنها صورت گرفته است، و این تغییر هنوز در Repo، ذخیره نشده است.

❖ **مشاهده ی محتویات:**

درحقیقت جهت مشاهده ی محتویات یک Repo در محیط Bash، از همان دستور ls استفاده می شود:

```
ls
```

اما مشکل این دستور این است که همه ی فایل ها را نشان نمی دهد، مثلن، فایل های مخفی و فایل ها با قالب خاص را لیست نمی کند.

پس بهترین دستور برای اطلاع از محتویات یک Repo، دستور زیر است:

```
ls -la
```

❖ **Clone a Repository:**

فرض کنید قصد داشته باشیم یک Repo که در جایی دیگر قرار دارد را، با تمام جزییات و تنظیمات و محتوا، در Directory فعلی که CL در آن قرار دارد، کپی کنیم.

ابتدا آدرس آن Repo را به دست می آوریم.

CL را به Directory موردنظر خود هدایت می کنیم و درج می کنیم:

```
Git clone آدرس مقصد
```

❖ درحقیقت گاهی یک Repo در یک سایت مثل GitHub، پس بطورمثال آدرس https آن را پیدا می کنیم و:

```
Git clone https://github.com/....git
```

❖ وقتی یک Repo، Branch های زیادی داشته باشد، Clone کردن آن مدت زمان زیادی نیاز خواهد داشت، و گاهی نیازی به همه ی Branch های گذشته ی آن نیست، و یا بالعکس، گاهی ما نیاز به یکی از Branch های گذشته ی آن Repo خواهیم داشت. پس می توانیم تنها یکی از Branch های آن Repo را Clone کنیم، مثلن فقط Master:

```
Git clone --branch=master https://github.com/....git
```

نکته: در بعضی Repo ها، نام Branch اصلی، main می باشد.

نکته: برای Branch های دیگر، کافی ست به جای master، نام آن Branch را درج کنیم.

نکته: اگر این دستور را به طور مثال به صورت زیر وارد کنیم، تنها سه Commit آخر آن را بارگذاری می کند:

```
Git clone --branch=master --depth=3 https://github.com/....git
```

❖ فرض کنید در پوشه ی User، یک Repo به نام A1 و یک پوشه ی معمولی به نام B1 وجود دارد، و ما قصد داریم درون پوشه ی B1 یک Clone از A1 داشته باشیم. پس CL را به داخل B1 هدایت کرده و درج می کنیم:

```
Git clone ../A1
```

نکات:

- درحقیقت، علامت .. به معنای یک گام از Directory جاری، عقب تر، می باشد.
- به Repository که روی سیستم جاری قرار دارد، Local Repo و به Repo ای که روی یک Server قرار دارد، Remote Repo گفته می شود.
- اگر در مورد کار با یک دستور تحت Git نیاز به کمک داشته باشیم، کافی ست آن دستور را درج کرده و در پایان از help-- استفاده کنیم، مثال:

```
Git status --help
```

و اگر به جای دو خط تیره، از یکی استفاده کنیم، در همان محیط ترمینال، پارامترها را لیست می کند.

Git Server:

درحقیقت به **Server** ای که مخصوص ذخیره سازی **Repository** ها درنظرگرفته می شود، **Git Server** گفته می شود، که **Remote Repo** ها درون آن ذخیره خواهند شد.
بطورمعمول همه ی سازمان های کوچک و بزرگ، نیاز به یک **Git Server** دارند.

وبسایت هایی هستند که در آنها می توان در محیط گرافیکی که دارند، **Repository** ایجاد کرد و با دیگران به اشتراک گذاشت و تغییر داد.

- این وبسایت ها، **Online Git Server** هستند.
- پس تمام **.Repo** هایی که روی یک **Git Server** (چه تحت شبکه و چه Online) قرار دارند، **Remote Repository** نامیده می شوند.

سایت های زیر، **Online Git Server** می باشند:

https://github.com
https://gitlab.com
https://bitbucket.org

این وبسایت ها به برنامه نویس ها اجازه می‌دهند که فایل‌های خود را **upload** کرده و با سایر توسعه‌دهندگان همکاری داشته باشند.
به این شکل که ابتدا یک حساب کاربری در آنها ایجاد کرده و سپس برای هریک پروژه، یک **Repository** ایجاد می کنیم.

- ایجاد **Repository** در این وبسایت ها، ساده و به خوبی مشخص است.
- بطورمعمول یک فایل به نام **README.md** به **Repo** اضافه می کنیم که در آن توضیحات یا اطلاعاتی در مورد آن **Repo** درج خواهیم کرد.
- یک **Repo** را می توان **public** تعیین کرد تا هرکسی که به حساب ما سر می زند، بتواند آن **Repo** را ببیند و یا به شکل **Private** که فقط افرادی که خودمان تعیین می کنیم بتوانند آن را ببینند.

: Work on files

وقتی که یک File را درون پوشه ی Repo قرار می دهیم یا ایجاد می کنیم:

- در ابتدا در حالت **Working Directory** قرار دارد، که تحت کنترل Git نخواهد بود.

یعنی اگر دستور git status را وارد کنیم، می بینیم یک بخش جدید اضافه شده است بنام untracked files (فایل های پیوست و ردیابی نشده)

- پس ابتدا توسط دستور زیر، (مثلن برای فایل Test1.txt) آن را به وضعیت **Staging Area** تغییر می دهیم، که شناسایی شود:

```
git add Test1.txt
```

اگر بیش از یک فایل باشند:

```
git add Test1.txt Test2.txt
```

حال اگر دستور git status را وارد کنیم، می بینیم که دیگر آن فایل، از نوع untracked نیست، اما آماده ی Commit شدن است.

درحقیقت این فایل به وضعیت Staging Area (Staged) تغییرحالت داده است.

- **حالت دیگری وجود دارد به نام modified ، که مشابه Untracked است، اما مربوط به فایل هایی است که از قبل در Repo بوده اند، اما تغییر کرده اند و این تغییر در Repo ذخیره نشده است؛ به همین جهت، آنها نیز وارد وضعیت Staging Area شده اند. پس برای ذخیره ی فایل های modified نیز، دستور git add را بکار می گیریم.**

نکته: پس چه یک فایل را برای اولین در Repo اضافه یا ایجاد کنیم، و چه یکی از فایل های موجود در آن را تغییر دهیم، در هردو حالت، آن فایل ها به حالت Working Directory می روند، که لازم است آنها را ابتدا stage کرد.

- **در نهایت، برای تغییر حالت آن فایل، از Staging Area ، به حالت modified، که در Repositry ثبت و ذخیره شود، وارد می کنیم:**

نکته: ضروری است که توسط سوییچ m- ، (commit message) یک متن راهنما نیز درج کنیم، و سپس در پایان نام فایل را درج می کنیم و اینتر …

```
git commit -m "This is Adding a Test file. Test1.txt"
```

حال اگر دستور git status را وارد کنیم، می بینیم که هیچ تغییری که اخیرن ثبت شده بوده باشد را نمایش نمی دهد.

نکات:

- درج صحیح و مشخص Commit message، خیلی خیلی مهم و حیاتی است، چراکه در آینده، جهت جستجو های خاص میان Commit ها، مفید خواهد بود.
- اگر نام فایل را در پایان سوییچ m- درج نکنیم، تمامی فایل های موجود در Staging area را commit خواهد کرد.
- وقتی فایلی commit شده را حذف می کنیم، این حذف را نیز باید commit کنیم.
- وقتی فایل commit شده ای را تغییرنام می دهیم، نیاز به دو commit است، یکی برای <u>افزودن</u> فایل با نام جدید، و یکی برای <u>حذف</u> فایل با نام قدیمی.

✓ **فرض کنید بخواهیم فایلی که به Staging Area رفته است را دوباره به Working Directory بازگردانیم. پس:**

```
git restore --staged Test1.txt
```

✓ **فرض کنید تعداد فایل هایی که ویرایش یا اضافه شده اند، خیلی بیشتر شوند. پس add کردن آنها زمانبر خواهد شد، به همین جهت، کافی ست به جای نام فایل ها، یکبار علامت نقطه را درج کنیم، پس تمامی فایل های جدید و فایل های ویرایش شده را وارد Staging Area خواهد کرد:**

```
git add .
```

گاهی به جهت کار با محیط Git از دو محیط متفاوت مثل cmd و Git Bash، ممکن است دستورهای add و commit از کار افتاده و خطای زیر ظاهر شود:

```
Another git process seems to be running in this repository, e.g.
an editor opened by 'git commit'. Please make sure all processes
are terminated then try again. If it still fails, a git process
may have crashed in this repository earlier:
remove the file manually to continue.
```

جهت رفع این مشکل، کافی ست یکبار دستور زیر را، در حالی که Command Line در Repo قرار دارد، وارد کنیم:

```
rm -f .git/index.lock
```

Diff :

- ✓ **فرض کنید پس از وارد کردن دستور git status، می بینیم که یک یا تعدادی فایل modified داریم.**

برای مشاهده کردن تغییرات انجام شده روی آن فایل ها، که آنها را از committed به modified تغییر داده است، درج می کنیم:

```
git diff
```

- **تغییرات را فایل به فایل لیست می کند، و برای نمایش تغییرات هریک فایل، ابتدا نام آن را به شکل زیر درج می کند:**

diff --git a/نام فایل در حالت قبلی b/نام فایل در حالت جدید

- **معنی سطری که با @@ شروع می شود، به شرح زیر است:**

@@ -1, تعداد سطر درحالت قبل +1, تعداد سطر در حالت جدید @@

- **مثلن اگر به صورت @@ -1,9 +1,5 @@ باشد، یعنی قبلن دارای 9 سطر بوده، و در نسخه ی جدید، دارای 5 سطر است.**
- **پس از سطری که با @@ شروع می شود، محتوای سطر به سطر فایل را نشان می دهد، اما:**

نکته: در Git bash، پس از وارد کردن این دستور، نتیجه را در یک محیط Editor مانند نشان می دهد که برای خروج از این محیط، کافی ست Q را بفشاریم.

- **در پشت سطرهایی که در محتوای آنها، متن یا فضای خالی اضافه شده است، علامت + درج می شود.**
- **در پشت سطرهایی که در محتوای آنها، متن یا فضای خالی، حذف یا دچار تغییر شده است، علامت - درج می شود.**

برای خروج از حالت diff، کافی ست یکبار دکمه ی Q را از روی کیبورد بفشاریم.

نکته: درحقیقت، دستور git diff تفاوت ها حالت فعلی و حالت HEAD را نشان می دهد. یعنی هر دو دستور زیر، یک معنی را می دهند:

Git diff
Git diff HEAD

- حال فرض کنید بخواهیم تفاوت و تغییرات میان حالت فعلی را با commit قبل از HEAD مشاهده کنیم:

Git diff HEAD^

- دریافت تفاوت ها و تغییرات میان حالت فعلی با دو commit قبل از HEAD:

Git diff HEAD^^

- دریافت تفاوت ها و تغییرات میان حالت فعلی با پنج commit قبل از HEAD:

Git diff HEAD~5

- برای اینکه فقط تغییرات یکی از فایل ها را (مثلن فایل Test1.txt) مشاهده کنیم:

Git diff HEAD~4 Test1.txt

- برای خروج از حالت diff، کافی ست یکبار دکمه ی Q را از روی کیبورد بفشاریم.

Gitignore :

✓ **فرض کنید درون پوشه ی Repo، فایل یا فایل هایی با یک فرمت خاص قرار دارند که نمی خواهیم در Repo ثبت و دیده شوند، حتی زمانی که همه ی فایل های untracked را Add می کنیم، آن فایل های موردنظر، همچنان Add و به Staging Area تغییر حالت ندهند، و با وارد کردن دستور git status، آن فایل ها را به عنوان ثبت نشده نشان ندهد.**
بطورمثال در برنامه نویسی Python تحت Git، نیازی به ثبت و افزودن پوشه ی _Pycashe_ به Repo نیست.
مثلن دراینجا برای فایل های با فرمت jpg.

❖ **ابتدا فایلی با نام gitignore را به صورت مخفی، در master** (فضای پوشه ی اصلی پروژه) **ایجاد می کنیم.**

```
Touch .gitignore
```

❖ **ابتدا فایل gitignore را باز کرده و خط زیر را در آن درج و آن را ذخیره کرده و می بندیم!**

*.jpg

❖ **حال اگر دوباره چک کنیم، می بینیم که آن فایل یا فایل های jpg. ، به عنوان untracked ظاهر نمی شوند، و تنها خود فایل gitignore به شکل ذخیره نشده ظاهر می شود. پس خود فایل gitignore را به سادگی commit می کنیم.**

نکته: برای ایجاد فایل gitignore در ویندوز، لازم است که برنامه ی Notepad را باز کرده و آن را یکبار درون پوشه ی Repo، با نام gitignore. ذخیره کنیم.

نکته: اگر بخواهیم تنها یک فایل از Git نادیده گرفته شود، بهترین کار این است که آن فایل را درون یک پوشه منتقل کنیم، و در gitignore، مسیر آن پوشه را درج کنیم. چون فرض کنید نام آن فایل را در gitignore اضافه کنیم، کافی ست نام خود آن فایل را تغییر دهیم، پس دیگر در gitignore ثبت نخواهد بود!

مثلن برای پوشه ای به نام my File، در فایل gitignore یک سطر اضافه کرده و درج می کنیم:

/my File/

نکته: از حال به بعد، هر پوشه و هر فایلی که درون این پوشه (که خود my File در master قرار دارد)، ایجاد شود یا تغییر داده شود، از دید Git نادیده گرفته می شود.

نکته: گاهی که یک فایل یا Directory را در gitignore اضافه کرده و سپس از آن حذف کرده و برای بار دوم آن را اضافه می کنیم، دیگر اثر ignore روی آن فایل یا Directory عمل نمی کند!

بهتر است دقت کنیم، که گاهی نیز بالعکس، جهت ignore کردن یک فایل، لازم است که ابتدا آن را از Repo حذف کرده، سپس نام و آدرس آن را در لیست gitignore قرار دهیم، و سپس آن فایل را دوباره به Repo اضافه کنیم، و خواهیم دید که از دید Git چیزی اضافه نشده است!

وقتی بر روی پروژه های مختلف کار می کنیم، یک سری فرمت های خاص را همیشه نیاز است که در gitignore. درج کنیم، چون نیازی به افزودن آن فایلها به پروژه ی Git نیست، طوری که فایل gitignore، یک لیست پیشفرض طولانی خواهد داشت.
پس وبسایت زیر، این لیست پیشفرض را، به سادگی در اختیار قرار می دهد، کافی ست نام زبان موردنظر را در کادر ورودی آن وارد کنیم، تا لیست gitignore را نمایش بدهد.

www.gitignore.io

Git rm :

درحقیقت در command line های Shell (تحت linux) **توسط دستور rm می توان به سادگی فایل ها را حذف کرد.**

اما در Git، توسط دستور git rm، علاوه بر حذف آن فایل، آن فایل را از تاریخچه ی Git نیز بطورصحیح و کامل حذف خواهد کرد.

```
Git rm NameOfFile
```

Git Clean :

- **این دستور، روی فایل های untracked** (فایل هایی که هنوز به Repo معرفی و در آن ثبت نشده اند) **و یا ignored کار می کند.**
- **عملکرد این دستور، مشابه عملیات undo است؛ به این معنی که، فایل های untracked ای که در مسیر Repository قرار دارند را حذف می کند.**
- **این دستور، بطورپیشفرض، روی پوشه ها کاری انجام نمی دهد.**
- **این دستور، فایل هایی که در gitignore. مشخص شده باشند را پاک نمی کند.**

- **این دستور، توسط یک سری پارامترها کار می کند. پیشنهاد می شود که قبل از استفاده از این دستور، ابتدا از پارامتر n بصورت زیر استفاده شود که در پاسخ، فایل هایی حذف خواهند شد را فقط نشان می دهد و هیچ عملیات دیگری انجام نخواهد شد:**

```
git clean -n
```

- **پس برای حذف کردن فایلهای untracked، از دستور زیر** (force) **استفاده می کنیم:**

```
git clean -f
```

- **برای اینکه حذف کردن فایل ها را بصورت interactive** (تعاملی/ نمایش درخواست تایید) **انجام دهیم، از دستور زیر استفاده می کنیم:**

```
git clean -i
```

پس در پاسخ، یک لیست ظاهر می شود که عملیات ها را لیست کرده و برای هریک یک شماره اختصاص داده است.

برای اعمال هریک از عملیات ها، کافی ست عدد آن را درج و اینتر کنیم.

جهت حذف تنها یکی یا برخی از فایل های untracked، کافی ست شماره ی گزینه ی select by numbers را وارد کنیم؛ و ... .

گزینه ی filter by pattern مشابه همان عملیات exclude است که یکی یا تعدادی یا یک سری از فایل ها را از فرآیند clean جدا می کند.

نکته: اگر در هریک از دو دستور بالا، به پارامترها، یک d اضافه کنیم، یعنی dn- و df- و di- ، پوشه ها نیز هدف عملیات قرار خواهند گرفت.

نکته: اگر در هریک از دو دستور بالا، به پارامترها، یک x اضافه کنیم، یعنی xn- و xf- و xi- ، فایل هایی که در لیست gitignore قرار دارند نیز هدف عملیات قرار خواهند گرفت.

نکته: برای اینکه هم پوشه ها و هم فایل های ignore شده را نیز در عملیات clean هدف قرار دهیم، پارامترها بصورت dxn- و dxf- خواهند شد.

Command and Query control :

Git Stash :

فرض کنید، در حین کار در Git یک مشکل یا کاری در جایی دیگر پیش می آید، در حالی که Repo را commit نکرده ایم و فعلن قصد commit کردن آن را نیز نداریم!
حال اینکه Repo را بدون commit کردن رها کنیم، زیاد منطقی و امن نیست! در اینجا، می توان Repo را توسط قابلیت Stash، به همان شکلی که هست، ذخیره سازی کرد تا بعد، دوباره به همان شکل، آن را ادامه داد.

پس فرض کنید، فایلی داریم به نام Test.txt، و متنی در آن وارد کرده و ذخیره اش می کنیم و git status می گیریم.
می بینیم که Test.txt به حالت modified تغییر پیدا کرده است. حال نمی خواهیم آن را commit کنیم، پس کافی ست یکبار دستور زیر را وارد کنیم:

```
Git stash
```

پس تغییرات اعمال شده، به شکل خودکار ناپدید شده و در حافظه ی Git ذخیره می شوند.
پس از اعمال این دستور، همه ی فایل های تغییریافته، به شکل گزینه های جدا، در Stash ذخیره می شوند.

کافی ست یکبار دستور git status را بزنیم، می بینیم که هیچ تغییری وجود ندارد! چون همه ی آنها به حافظه ی stash منتقل شده اند.
توسط دستور زیر می توان، فایل هایی که در حافظه ی stash ذخیره شده اند را مشاهده کرد:

```
Git stash list
```

هر فایل حاضر در Stash، دارای یک ID است. در پاسخ ظاهرشده، ID هر فایل، درون {ID}@ درج می شود.
برای مشاهده ی جزییات بیشتر فایل های منتقل شده به انبار Stash، دستور زیر را وارد می کنیم:

```
Git stash show
```

نکته: تنها فایل هایی که در Staging area هستند، می توانند به Stash ارسال شوند، در غیراینصورت، همچنان untracked می مانند؛ مگر اینکه دستور زیر را وارد کنیم که همه ی تغییرات را به اضافه ی فایل های untracked، به stash ارسال خواهد کرد:

```
Git stash -u
```

درحقیقت، وقتی از انتقال تغییرات به Stash مطمین شدیم، حال می توانیم تغییرات جدیدی اعمال و حتی commit کنیم، و سپس دوباره آن تغییرات را از Stash فراخوانی کنیم.

- ✓ **اگر دستور زیر را وارد کنیم، <u>تنها فایلی که Stash ID آن را وارد کردیم</u>، بازگردانی می شود:**

```
Git stash apply ID
```

- ✓ **اگر دستور زیر را وارد کنیم، <u>همه ی فایل های موجود در Stash</u>، بازگردانی می شوند:**

```
Git stash apply
```

نکته: درحقیقت وقتی محتویاتی را با Stash منتقل کنیم، آن فایل ها، پس از بازگردانی، همچنان در حافظه ی stash باقی می مانند، پس توسط دستور زیر، Stash list را خالی می کنیم.

```
Git stash clear
```

با توجه به اینکه ما می توانیم بارها تغییراتی را روی Repo اعمال کرده و هریک از مجموعه تغییرات را به Stash ارسال کنیم، و از طرفی می دانیم که در هربار اعمال دستور stash، فایل ها به شکل جداجدا و با ID های متفاوت ذخیره می شوند، بهتر است که هربار یک مجموعه تغییرات را به Stash ارسال می کنیم، آن را به شکل یک مجموعه تغییر و با یک نام برای همه ی مجموعه ی فایل ها در آن حالت، تعریف کنیم؛ پس می توان اعمال Stash را نیز به صورت زیر انجام داد:

```
Git stash save "نام دلخواه برای استش"
```

- ✓ **پس با هربار فراخوانی دستور git stash list، تنها لیست هریک مجموعه تغییر ذخیره شده را می بینیم.**
- ✓ **پس می بینیم که هریک از مجموعه تغییرات ذخیره شده، علاوه بر نام دلخواهی که تعیین کرده ایم، یک ID هم خواهد داشت.**
- ✓ **بازگردانی Stash ها به همان حالت معمول است.**

: Git log

❖ جهت مشاهده ی مشخصات Commit هایی که در Git بر روی همه ی تک تک فایل ها انجام شده است، کافی ست از دستور زیر استفاده کنیم:

```
Git log
```

می بینیم که برای هر Commit:

- ➢ یک ID به شکل یک کد رمزنگاری شده ی 40 بیتی اختصاص می دهد، که هر Commit یکتا بماند، به آن Commit ID گفته می شود.
- ➢ نام Author یا اعمال کننده ی آن Commit را نشان می دهد.
- ➢ تاریخ و ساعت دقیق اعمال آن Commit را نشان می دهد.
- ➢ Commit message را نشان می دهد.

نکته: به خوبی می بینیم که در مقابل آخرین Commit انجام شده، عبارت (HEAD -> main) درج خواهد بود.

نکته: پس از وارد و اجرا کردن دستور log، بطورخودکار به CL بر نمی گردد، پس برای این کار، از دکمه ی Q روی کیبورد استفاده می کنیم.

✓ **همچنین می توان دستور زیر را وارد کرد، که فقط شامل commit message ها و هفت حرف/رقم اول ID است:**

```
git log --oneline
```

✓ پس وقتی دستور git log --oneline را وارد می کنیم، در سطر اول:

```
xxxxxxx (HEAD -> CurrentBranch, BranchNames) LastCommitMessage
```

✓ **اگر به پایان دستور git log، گزینه ی -graph - را اضافه کنیم، توسط یک سری علایم در پشت مشخصات Commit ها، تغییرات را در Branch های مختلف، به زیبایی تفکیک می کند.**

❖ **جهت مشاهده ی تغییرات اعمال شده بر روی یک فایل مشخص، پس از ثبت هریک از Commit ها کافی ست از دستور زیر استفاده کنیم:**

Git blame نام آن فایل

: Git tag

می دانیم که توسط دستور git log، تنها چیزی که معرف هریک از Commit هاست، شماره ID چهل بیتی آنهاست (Commit ID)؛ که بخاطرسپردن آنها، کار ساده ای نیست و مدیریت، کنترل و مانیتور کردن Commit ها را پیچیده و سخت می کند.
درحقیقت، برای Commit message هم که نیاز است که فقط توضیحات مربوط به وضعیت آن Commit درج شود.
پس بهترین کار این است که به هریک از Commit ها، یک برچسب شناسایی بزنیم که با نام Tag آن را می شناسیم.

- **برای اینکه قبل از ثبت یک Commit، به آن یک Tag اختصاص دهیم، کافی ست ابتدا توسط دستور زیر، یک Tag برای آن ایجاد کنیم:**

  Git tag "یک نام یا شماره ی شناسایی دلخواه"

  حال اگر یکبار Commit کرده و تغییرات فعلی را ثبت کنیم، و دستور git log را وارد کنیم، می بینیم که در انتهای ID مربوط به Commit ثبت شده، مقدار tag ای که اختصاص داده ایم نیز درج شده است.

- **اما گاهی لازم است که به Commit هایی که در گذشته هم ذخیره کرده ایم، یک Tag اختصاص دهیم؛ پس کافی ست توسط همان دستور git log، شماره ی ID مربوط به Commit موردنظر را پیدا و کپی کرده و توسط دستور زیر، به هریک از Commit های قدیمی هم یک Tag اختصاص دهیم:**

Git tag "یک نام یا شماره ی شناسایی دلخواه" ID

Branch :

همانطور که می دانیم، توسط ایجاد کردن Branch ها:

- **می توان روی Repo، یک سری عملیات موردنظر را تست و بررسی کرد و در صورت ناموفق بودن عملیات، همه ی عملیات را حذف کرد.**
  به این شکل که، ابتدا از Repo اصلی، یک Branch تهیه کرده و تغییرات را روی آن اعمال می کنیم.
- **می توان از کمک کاربرهای دیگر، در پیشبرد پروژه استفاده کرد.**
  به این شکل که برای هرکاربر که باید یک نسخه از پروژه را دریافت کند، یک Branch ایجاد می کنیم.
  هر کاربر، پروژه ی خودش را (Branch خودش را) پیش می برد و در پایان، همه ی کاربرها، Branch هایشان را با Master، ادغام می کنند.

❖ مشاهده ی Branch:

جهت مشاهده ی لیست Branch ها و اطلاع یافتن از اینکه Repo جاری، تحت کدام Branch کار می کند:

```
Git branch
```

جهت مشاهده ی لیست همه ی Branch ها (چه local و چه Remote):

```
Git branch -a
```

❖ ایجاد یک Branch:

جهت ایجاد یک Branch جدید، از هریک از دو دستور زیر می توان استفاده کرد:

```
git branch NAME
git checkout -b NAME
```

مزیت ایجاد Branch توسط دستور checkout این است که بطورخودکار CL را نیز بر روی آن Branch تغییر می دهد.

❖ جهت تغییر Command Line به Branch اصلی:

```
git checkout Main
```
و یا
```
git checkout Master
```

❖ جهت تغییر Command Line به یک Branch موجود دیگر:

```
git checkout NAME
```

نکته: فرض کنید، دو Branch ایجاد کرده اید، در هر یک، تغییرات متفاوتی را بر روی یک فایل txt. اعمال می کنید.
با هربار switch کردن میان Branch ها، محتویات فایل ها نیز به شکلی که آنها را برای هر Branch ذخیره (commit) کرده بودیم، سوییچ می شوند (تغییر می کنند).

نکته: فرض کنید در یک Branch یک فایل را تغییر می دهید.

- حال بدون اینکه آن تغییر را staged و commit کنید، command Line را به یک Branch دیگر تغییر می دهید.
- درحقیقت، تغییراتی که در Branch قبل commit نشده بودند، ناپدید نمی شوند، بلکه در command Line جدید، همچنان آماده ی commit شدن هستند.
- حال اگر در Branch دوم، commit انجام دهیم، تغییراتی که در Branch قبل اعمال کرده، اما commit نکرده بودیم، برای این Branch دوم commit می شوند!

❖ جهت حذف یک Branch:

```
git branch -d NAME
```

درحقیقت، گاهی که تغییرات یک Branch را اصلن ذخیره یا در Branch های دیگر ادغام نکرده ایم، پس از وارد کردن این دستور جهت حذف آن Branch،
خطا می دهد> error: The branch is not fully merged.
پس از دستور زیر استفاده می کنیم:

```
git branch -D NAME
```

❖ جهت تغییرنام یک Branch:

```
git branch -m oldNAME NewName
```

❖ مشاهده ی log:

وقتی دستور زیر را وارد می کنیم:

```
git log --oneline --all
```

برای هر Branch فعال، ابتدا نام آن را همراه با آخرین Commit message ش درج، و تمام Commit های آنرا، در زیر آن لیست می کند:

```
xxxxxxx (BranchName) LastCommitMessage
```

اگر آخرین Commit برای دو یا چند Branch، یکسان باشد، پس اولین سطر آنها را در یک سطر خلاصه می کند:

```
xxxxxxx (BranchName1, BranchName2, BranchName3) LastCommitMessage
```

اگر Command Line جاری، در یکی از Branch های غیر master باشد، پس سطر اول برای آن Branch به شکل زیر خواهد بود:

```
xxxxxxx (HEAD, BranchName) LastCommitMessage
```

اگر Command Line جاری، در master Branch باشد، پس سطر اول برای master Branch به شکل زیر خواهد بود:

```
xxxxxxx (HEAD -> master) LastCommitMessage
```

برای مشاهده ی Log، از دستور زیر نیز می توان استفاده کرد:

```
git log --oneline --graph --all
```

Commit :

Amend :

فرض کنید یکی از فایل ها را تغییر داده ایم، اما قصد نداریم برای همان یک تغییر، یک Commit جدید ثبت کنیم، پس می توان این تغییر را به آخرین Commit ثبت شده، اضافه کنیم، همراه با Commit message ای که می تواند متفاوت با Commit message قبل باشد.

ابتدا تغییر یا تغییرات جدید را Add می کنیم، سپس آنها را به شکل زیر commit می کنیم که با آخرین Commit ثبت شده، یکی شوند:

```
git commit --amend -m "xxx"
```

پس به سادگی می توان:

- **Commit message ثبت شده را تغییر داد.**
- **برخی از تغییرات اعمال شده پس از ثبت یک Commit را، بدون یک ایجاد یک Commit جدید، به حالت قبل برگرداند.**
- **همه ی فایل های Untracked را حذف کنیم.**

نکته: این دستور تغییر را، بر روی Commit های که خودمان ثبت کرده ایم (بر روی نسخه ی local) اعمال می کنیم، و نه روی Commit های دیگران

Checkout :

فرض کنید بخواهیم به یکی از commit هایی که در گذشته اعمال کرده ایم، برگشته و وضعیت را ببینیم و دوباره به آخرین Commit اعمال شده، بازگردیم.

پس ابتدا توسط git log، مقدار commit ID موردنظر را پیدا کرده و مطابق دستور زیر وارد می کنیم:

```
git checkout xxxxxxx
```

پس اولین سطر در پاسخ، به شکل زیر خواهد بود:

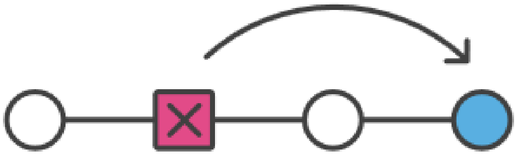
```
Note: checking out 'xxxxxxx'.
You are in 'detached HEAD' state. You can look around, make experimental
```

- **حال اگر به پوشه ی Repo نگاه کنیم، می بینیم که همه چیز در آن، به آن Commit بازگشته است.**
- **اگر دستور git log را وارد کنیم، می بینیم که Commit های بعدی، نمایش داده نمی شوند، اما در حقیقت از بین نرفته اند.**
- **اگر دستور git branch را وارد کنیم، می بینیم که درحقیقت ما در یک Branch هستیم، که در آن حالت در Commit گذشته بازگشته است.**
- **پس برای بازگشت به آخرین Commit ای که ایجاد کرده بودیم، کافی ست Branch فعلی را به Master یا Branch موردنظر، تغییر دهیم.**
- **پس ما می توانیم، با دستور Checkout، به هریک از Commit های گذشته بازگشته و یک Branch جدید ایجاد کنیم؛ چون اگر فقط با دستور checkout به یک commit بازگشته و تغییراتی اعمال کنیم، به محض اینکه به یکی از Branch ها بازگردیم، همه ی آن تغییرات محو می شوند.**

Git revert :

توسط این دستور، می توانStatus را به Status یکی از commit های قبلی بازگرداند!
این دستور زمانی که یک Bug را پیگیری می کنید و متوجه می شوید که توسط یکی از Commit های قبلی ایجاد شده است، کاربرد دارد.
پس به جای اینکه به صورت دستی Bug را برطرف کنیم و یک Commit جدید انجام دهیم، می توان از git revert استفاده کرد.
این دستور، آن commit موردنظر را همراه با تمامی تاثیرهایی که در Repo داشته است (شامل ایجاد یا حذف فایل و متن)، **حذف می کند.**

درحقیقت در این فرآیند، Commit موردنظر در گذشته را بطورکل، بی اثر کرده و تاثیرات آن را از بین می بریم و این بی اثری را به شکل یک commit جدید در زمان حال، ایجاد می کنیم.



- **ابتدا با دستور git log، مقدار ID آن Commit را پیدا می کنیم.**
- **سپس دستور زیر را وارد می کنیم:**

```
Git revert CommitID
```

- **سپس نیاز به یک متن، تاییدیه جهت ثبت یک commit برای revert موردنظر می باشد، که خودش یک متن پیشفرض را پیشنهاد می دهد. جهت تایید این متن و عملیات Revert، کافی ست یکبار** Esc **را زده، و مقدار** :wq **را درج و اینتر را بزنیم.**

نکته: درحقیقت دستور git revert، شباهت زیادی به کاربرد دکمه های میانبور Ctrl+Z و Ctrl+Shift+Z در کامپیوتر دارد.

نکته: تا جای ممکن، سعی می کنیم از این دستور و قابلیت، فقط بر روی آخرین Commit ها استفاده کنیم، چون هرقدر به Commit های قدیمی تر برگردیم که حذف و تغییرات اساسی تر فایل ها وجود داشته است، امکان بروز خطا در revert کردن بیشتر خواهد بود.
این خطا، با error: could not revert نشان خواهد داد، و می بینیم که نام Branch را در پایان CL ، همراه با REVERTING | نشان می دهد.
درنتیجه عملیات revert ناموفق بوده است، اما اگر دستور git status را بزنیم، می بینیم که برخی تغییراتی که ایجاد تداخل کرده اند را به شکل unmerged یا untracked لیست می کند.
پس جهت صرف نظر کردن از این revert و رفع این تداخل، از دستور زیر استفاده می کنیم:

```
git revert --abort
```

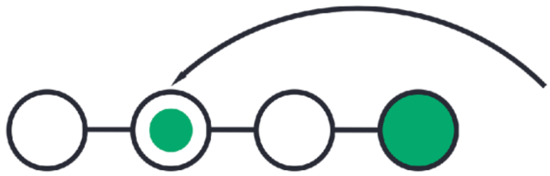
نکته: پس از انجام revert، تاریخچه ی Commit ها همچنان وجود خواهد داشت.

درحقیقت، با توجه به اینکه توسط Revert، هیچ commit ای حذف نخواهد شد، بلکه تاثیر آن را از بین برده و این بی تاثیری را به شکل یک commit جدید ثبت می کنیم، پس اگر خود یک revert commit را revert کنیم، آن revert را از بین خواهیم برد و دوباره به حالت قبل باز می گردیم!

: Git reset

با استفاده از این دستور، می توان repository را به یکی از commit های قبلی بازگرداند، طوری که هر تغییری پس از آن commit ، از بین خواهد رفت.

ابتدا باید Commit ثبت شده ی موردنظر را توسط دستور git log پیدا کنیم.



و سپس به سوی آن، Reset را انجام دهیم.



```
Git reset CommitID --soft
```

- **پس از انجام این reset، اشاره گر Head، به آن Commit موردنظر در گذشته تغییر خواهد کرد، اما داده هایی که در طی Commit های بعدی ایجاد شده بوده اند، حذف نشده و تنها به حالت Staging می روند. که به سادگی می توان آنها را دوباره بازگرداند.**

- **اگر از دستور زیر استفاده کنیم، اشاره گر Head، به آن Commit موردنظر در گذشته تغییر خواهد کرد و داده هایی که در طی Commit های بعدی ایجاد شده بوده اند، حذف نشده و تنها به حالت به حالت untracked می روند.**

```
Git reset CommitID --mixed
```

- **اما اگر از دستور زیر استفاده کنیم، هم در تاریخچه ی Git log، تمام Commit هایی که پس از Commit موردنظر ایجاد شده بوده اند، و همچنین تمام داده هایی که در طی آن Commit های بعدی، ایجاد شده بوده اند نیز حذف خواهند شد:**

```
Git reset CommitID--hard
```

نکته: پس از اعمال هریک از این سه حالت دستور Reset، هیچیک از commit هایی که پس از آن commit موردنظر بوده اند، قابل بازیابی نخواهند بود.

Git rebase :

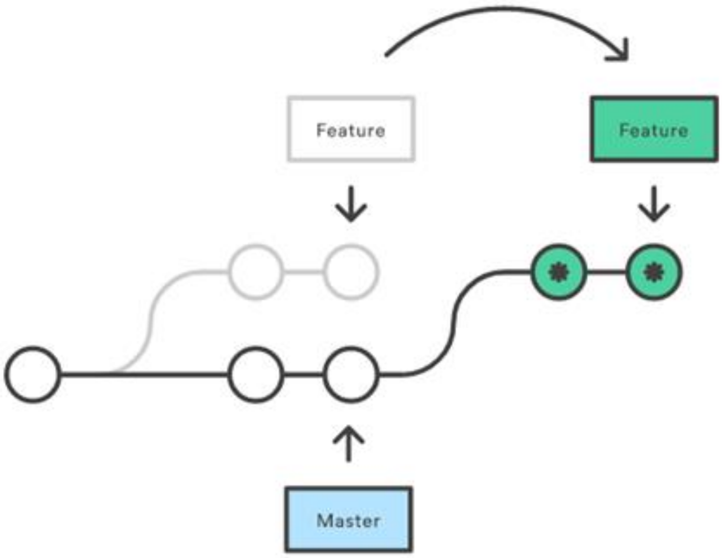
واژه ی Rebase ، به معنی تغییر دادن پایه یا منشا و منبع است.

از این دستور، به دو شکل کلی می توان استفاده کرد:

1. فرض کنید یک Branch ایجاد می کنیم، به نام Feature.

سپس در طول زمان، در Master Repo چند Commit ثبت می شود.

حال می خواهیم آن Branch ای ایجاد کرده بوده ایم نیز، آن تاثیراتی که در طی Commit های بعدی ایجاد شده اند را نیز دریافت کند؛ طوری که انگار Branch را پس از آخرین Commit حال حاضر ایجاد کرده ایم.



پس برای rebase کردن یک Branch:

- ابتدا وارد آن Branch می شویم.
- دستور زیر را وارد می کنیم:

```
git rebase master
```

.2

- می دانیم که گاهی تعداد Commit ها و تغییرات اعمال شده، میان آن Commit ای که Branch از آن ایجاد شده است تا آخرین commit ای که master اکنون در آن قرار دارد، زیاد است، طوری که نیازی به برخی یا خیلی از تغییرات نیست، پس لازم است rebase را طوری اعمال کنیم که برخی تغییرات یا Commit ها نادیده گرفته شوند.
- گاهی در همان Branch ای که هستیم، لازم است که یکی از commit های قبلی را حذف و تاثیر آن را از بین ببریم.
- گاهی لازم است که برخی از commit های قبلی را با هم ترکیب و تبدیل به یک commit کنیم.

برای هریک از این حالات، از دستور rebase به شکل interactive استفاده می کنیم. مثلن توسط دستور زیر، هر سه commit قبل از Head را فراخوانی می کنیم تا روی آنها کار کنیم:

```
git rebase -i HEAD~3
```

وارد ویرایشگر متنی VIM شده که هر سه Commit آخر را لیست می کند.

نکته: درحقیقت ترتیب نمایش commit ها در این ویرایشگر، به ترتیب از قدیمی ترین commit در بالاست که جدیدترین commit را در پایان این لیست خواهیم داشت.

- ✓ **نوع عملیاتی که قرار است بر روی هریک انجام گیرد را با درج یک کلمه در پشت هر Commit مشخص می کنیم.**
- ✓ **لیست و معنی هر کلمه را در لیستی که در پایین قرار دارد، می بینیم.**
- ✓ **بطورپیشفرض، کلمه ی pick قرار دارد، که به معنی نگه داشتن آن commit بدون هیچ تغییری می باشد.**
- ✓ **می بینیم که در ابتدای هریک از آن کلمات، یک حرف در پشت کاما قرار دارد، به این معنی است که می توان CL را پشت هر commit برده و آن حرف را درج و یکبار اینتر را زد تا اعمال شود.**

نکته: اما باید دقت کرد که در این حالت، جهت اعمال حرف موردنظر بر روی یک Commit، لازم است ابتدا یکبار وارد text mode شده و یک سطر فاصله میان آن commit با commit بعدی ایجاد کرد و دوباره به command mode برگشت و حرف موردنظر را اعمال کرد. چون در این حالت، عملیاتی که دستور حرفی اعمال می کند، تا یک سطر بعد هم اعمال می شود!

عملیات squash: با درج این عبارت در پشت یک commit، آن را با یک commit قبل از خودش **ترکیب** می کند و سپس درخواست درج یک نام برای نتیجه ی ادغام شده می دهد. **(که بطورپیشفرض نام همه ی commit های ادغام شده را پیشنهاد می دهد.)**
درصورت نیاز به انجام squash بر روی بیش از دو commit، لازم است که در پشت همه ی آنها، عبارت squash را درج کنیم.

نکته: نمی توان در پشت اولین commit در این لیست، squash درج کرد، چرا که نمی توان این عملیات را بر روی commit هایی که وارد عملیات rebase نکرده ایم اعمال کنیم!

پس از اعمال تغییرمورد نظر و ذخیره و خروج از محیط ویرایشگر، یکبار git status می گیریم.

در صورت بروز [Conflict](#)، آن را تصحیح و رفع کرده و add و commit می کنیم.

درصورت پیچیده شدن rebase، می توان آن را توسط دستور زیر کاملن لغو کرد:

```
git rebase --abort
```

- ➢ **پس از رفع conflict، جهت اتمام فرآیند rebase، کافی ست دستور زیر را وارد کنیم:**

```
git rebase --continue
```

- ➢ **درصورت موفقیت آمیز بودن rebase، پیام زیر را در پاسخ دریافت خواهیم کرد:**

```
Successfully rebased and updated refs/heads/M-new.
```

Git cherry-pick :

فرض کنید در یک Repo چندین Branch وجود دارند.
در هر Branch، محتویات متفاوتی می توانند موجود باشند.
اما یک فایل در درون همه ی Branch ها وجود دارد. مثلن فایلی به نام Test.txt
یکی از کاربرها که فقط با یکی از Branch ها کار می کند، متوجه می شود که باید لازم است بخشی از فایل Test.txt برای همه یا برخی از Branch های دیگر تغییر کند.
پس این کاربر در Branch خودش آن تغییر موردنظر را اعمال و آن را با یک Commit ثبت می کند. مثلن یک Commit به نام "change done"
حال برای اینکه این تغییر بر روی فایل Test.txt مربوط به هریک از Branch های دیگر نیز اعمال شود، کافی ست کاربرهای branch های دیگر، commit ID مربوط به "change done" **را دریافت کرده و در Branch خودشان، از دستور** cherry-pick **به شکل زیر استفاده کنند:**

```
git cherry-pick xxxxxxx
```

Github:

- بزرگترین Host برای میزبانی پروژه‌های توسعه دهندگان (برنامه نویس ها) محسوب می‌شود که توسط یک شرکت آمریکایی ایجاد شده است.
- هم به صورت رایگان برای پروژه‌های عمومی، و هم بصورت پولی برای پروژه‌های تجاری قابل استفاده است.
- یک شبکه‌ی اجتماعی نیز محسوب می‌شود. پس امکاناتی همچون Follow کردن کاربران، قابلیت نظر دادن و لایک را نیز دارا می باشد.

پس از ورود به یک Repo، در تب Code ، لیست محتویات آن Repo را می بینیم.

- با کلیک بر روی هر File می توان تنظیمات و محتوای آن را مشاهده کرد.
- توسط دکمه ی Add file می توان به سادگی یک فایل را upload کرد و یا یک فایل متنی ایجاد کرد.
- در بالا سمت راستِ کادر میانِ صفحه، تعداد commit ها درج می شود که با کلیک بر روی آن، به سادگی می توان به صفحه ی لیست Commit ها رفت.

- توسط گزینه ی History می توان تاریخچه ی Commit ها را مشاهده کرد.
- توسط آیکون قلم (Edit) می توان محتوای آن را تغییر داد.
- توسط دکمه ی Raw می توان محتوای داده را به شکل یک صفحه ی مجزا و نهایی مشاهده کرد.
- توسط دکمه ی Blame، در پنجره ی باز شده، در سمت راست، محتوا را می بینیم و در سمت چپ، آخرین Commit ای که روی آن فایل، تغییر اعمال کرده است را، توسط Commit message اش، می بینیم.
  - با کلیک بر روی هریک از commit message ها، گزینه های زیر را خواهیم دید:
    - فایل های موردتغییر واقع شده را،
    - تغییرات اعمال شده بر روی هریک از فایل ها را،

نکته: این لیست تغییرات، به دو شکل ظاهری، قابل نمایش خواهند بود: unified و split .

Branch in Github:

اگر در تنظیمات یک Repo، وارد view all Branches شویم، لیست Branch ها را می بینیم و هم می بینیم که Default Branch (معمولن) master یا main می باشد، و هم می توانیم یک Branch جدید ایجاد کنیم.

برای تغییر default branch، کافی ست وارد settings آن Repo شده و به بخش Branches رفته و در آنجا هم می توان Branch موردنظر را تغییرنام داد، و هم می توان default Branch را تغییر داد.

Issue:

در تب Issues، می توان:

- برای Repo های دیگران، نقد و نظر و پیشنهاد درج کرد؛
- نقد و نظرات دریافتی برای Repo خودمان را بررسی و مدیریت کنیم؛ و حتی در صورت نیاز به اقدام برای یک Issue دریافتی، در همانجا می توان بطورمستقیم یک Branch برای آن Issue ایجاد کرد.
- Issue ها مثل یک تشکیل پرونده، هستند که باید به آنها رسیدگی و بالاخره بسته شوند.

Make a Fork:

گاهی در Github یک Repo از یک کاربر دیگر پیدا می کنیم که قصد داریم آن را تصحیح کنیم و یا در تکمیل آن همکاری کنیم. پس در صفحه ی اصلی آن، در بالا، بر روی Fork کلیک کرده و با کلیک روی Create fork، یک نسخه از آن، در حساب Github ما، بارگذاری می شود که می توان روی آن کار کرد.
سپس جهت کار بر روی این Repo جدید، عاقلانه ست که ابتدا یک Branch جدید در آن ایجاد کرده و بر روی آن تغییراتمان را اعمال کنیم.
پس از commit کردن تغییرات موردنظر، بطورخودکار پیغامی در بالای این Repo ظاهر می شود [Compare & pull request] که اگر روی آن کلیک کنیم و پیام Able to Merge را بدهد، پس می توان به سادگی یک تیتر و یک متن برای صاحب اصلی آن Repo ارسال کرد، تا تغییرات جدید را برروی نسخه ی اصلی اضافه کنیم. سپس صاحب Repo اصلی، در حساب خودش، در تب pull requests، این درخواست را دریافت می کند، و می تواند با باز کردن آن Request و کلیک بر روی Mergae pull request، آن را تایید کند. حال اگر صاحب Repo اصلی، این درخواست را تایید کند، تغییرات اعمال شده، به شکل مستقیم بر روی نسخه ی Master آن Repo که از آن Fork گرفته بوده ایم نیز، اعمال می شوند.

Connect to a Git Server :

درحقیقت نیازی نیست تا پرسنل [IT] هربار، به Git Server یک Remote Session برقرار کنند، تا وارد Remote Repo موردنظر شده و تغییرات موردنظرشان را اعمال کنند.

اولن: روش کار به این شکل است که در هربار نیاز به کار بر روی یک Repo موجود روی Git Server:

1. **کافی ست از آن Repo، یک Clone دریافت کنیم،**
2. **تغییرات را روی نسخه ی Clone شده، اعمال کنیم،**
3. **و در پایان نسخه ی جدید را بر روی Server ارسال (push) کنیم.**

دومن: دو روش معروف، جهت اتصال به Git Server وجود دارد:

- **توسط پروتکل https**
- **توسط پروتکل SSH**

نکته: بطورمعمول، اتصال به Git Server را توسط پروتکل SSH انجام می دهیم، که رمزنگاری قدرتمندی دارد، و پس از راه اندازی آن، دیگر نیازی به وارد کردن Credential در هربار اتصال نخواهد بود.

- پروتکل ssh باید از قبل، روی سیستم (چه ویندوز و چه لینوکس) نصب باشد. برای بررسی نصب آن، کافی ست در cmd یا در Terminal کلمه ی ssh را درج و اینتر کنیم که نباید خطا بدهد.
- همچنین می توان از وجود و نصب بودن سرویس مخصوص پروتکل SSH، یعنی Open SSH مطمین شد، به اینصورت که باید فایل های آن در مسیر C:\Windows\System32\OpenSSH موجود باشند.

1. **ابتدا لازم است که آن کاربر، توسط کامپیوترش که به شبکه متصل است، یک Certificate ایجاد کند.**

این Certificate شامل دو کلید خواهد بود؛ یک کلید عمومی (Public Key) و یک کلید خصوصی (Private Key).

این Key ها را می توان توسط ابزارهای مختلفی ایجاد کرد.

- **در cmd و یا در Git Bash (ویندوز) و یا در Terminal (لینوکس):**

  - **در Terminal دستور زیر را وارد می کنیم:**

ssh-keygen

اولین سوالی که می پرسد: درخواست تایید محل ذخیره ی این کلیدها خواهد بود.

درحقیقت مسیر پیشنهادی، مسیری است که با نام Keychain path شناخته می شود، و در Linux، بطورپیشفرض ~/.ssh/ و در Windows بطورپیشفرض C:\Users\USERNAME\.ssh می باشد.

اگر این مسیر و یا نام فایل را (که معمولن id_rsa است) تغییر دهیم، در آینده، درهربار اتصال به Git Server، باید، Private Key را از نو، به Git معرفی کنیم، که درصورت نیاز مداوم به اتصال به Git Server، طاقتفرسا خواهد بود؛ مگر اینکه در Git Server، این آدرس و یا نام را به شکل پیشفرض تغییر داده باشیم.

سوال دوم: می توانیم یک رمزعبور به نام Passphrase هم تعیین کنیم که در هربار استفاده، نیاز به وارد کردن آن نیز باشد.

سوال سوم: درخواست تایید مجدد Passphrase است؛ که اگر آن را خالی رها کرده بودیم، اینبار هم خالی رها کرده و اینتر و تمام.

نکته: Key ایجاد شده شامل دو فایل با نام id_rsa (کلید خصوصی) و id_rsa.pub (کلید عمومی) خواهد بود؛ که این Public Key را می توان توسط Notepad هم باز کرده و خواند.

- **و یا توسط ابزار PuTTYgen** (که با نصب PuTTY در ویندوز، افزوده خواهد شد):

❖ **این نرم افزار را باز کرده و روی Generate کلیک کرده و جهت پیشرویِ فرآیند، موس را روی صفحه ی آن حرکت می دهیم!**
❖ **در پایین این نرم افزار، گزینه ی ECDSA را انتخاب و از گزینه ی کرکره ای پایین، گزینه ی nistp384 انتخاب می کنیم.**
❖ **دوباره روی Generate کلیک می کنیم. جهت پیشرویِ فرآیند ایجاد کلید، موس را روی صفحه ی آن حرکت می دهیم!**
❖ **سپس، برای نوار Key comment، آدرس ایمیل را وارد می کنیم.**

❖ **برای گزینه ی passphrase و تکرار آن، یک رمزعبور نه چندان پیچیده را وارد می کنیم.**
❖ **بر روی دکمه های Save public Key و Save private Key به شکل جداگانه کلیک کرده و آنها را ذخیره می کنیم.**
نکته: فایل private key بطورپیشفرض، دارای پسوند ppk. خواهد بود.

نکته: کلید خصوصی را در سیستم خود نگه داشته و به هیچ وجه آن را با کسی اشتراک گذاری نکرده و به کسی ارسال نمی کنیم.

نکته: درحقیقت هرکجا یا در هر Server ای که نیاز به یک SSH Key باشد، که معمولن دکمه ی شبیه به Add SSh key وجود خواهد داشت، در آنجا لازم است که فایل id_ecdsa.pub را یکبار باز کرده و فقط خط اول را کامل کپی و paste کنیم.

2. باید کلید عمومی آن کاربر، درون Git Server ثبت شود.

در Github.com:

پس از ورود با حساب کاربری، وارد Settings شده و از سمت چپ، وارد SSH and GPG keys شده و در مقابل SSH keys، روی New SSH key کلیک کرده و گزینه های زیر را پر می کنیم:

- **Title**: یک نام دلخواه و مرتبط
- **Key Type**: Authentication Key
- **Key**: تمام محتویات فایل Public Key را کپی و در اینجا paste می کنیم.

3. باید کلید خصوصی آن کاربر، به صورت local ثبت شود.

➢ **در ویندوز:**

- **ابتدا باید حق دسترسی به فایل Private Key را تنها به خودمان (Client موردنظر) اعطا کنیم و نه هیچکس دیگر.**
- **PowerShell را با دسترسی Admin باز کرده و به ترتیب سه دستور زیر را وارد می کنیم تا اطمینان حاصل کنیم که سرویس ssh-agent که مسیول کنترل SSH Key هاست، در حال اجراست و با هربار reboot کردن سیستم، بطورخودکار اجرا خواهد شد:**

```
Get-Service ssh-agent | Set-Service -StartupType Automatic
Start-Service ssh-agent
Get-Service ssh-agent
```

❖ **سپس به Git Bash** (ویندوز) **و یا Terminal** (در لینوکس) **برگشته و:**

ابتدا command Line را به محل ذخیره سازی Key ها می بریم. سپس توسط دستور زیر، private key را معرفی می کنیم:

ssh-add نام فایل کلیدخصوصی

نکته: درصورت موفقیت آمیز بودن معرفی آن، خطی مشابه Identity added ظاهر می شود، درغیر اینصورت، ابتدا دستور زیر را وارد می کنیم، و دوباره دستور ssh-add را امتحان می کنیم:

eval $(ssh-agent -s)

4. **جهت اینکه کاربر بتواند توسط سیستم خودش به Git Server متصل شود:**

- **توسط Git Bash و یا CMD: دستور زیر را وارد می کنیم:**

SSH *FQDN-of-GitServer*

مثلن برای اتصال به Github:

```
ssh git@github.com
```

- **اولین کاری که پس از اتصال موفقیت آمیز با Git Server انجام می دهیم، clone گرفتن از Repo موردنظر است. مثال:**

```
Git clone FQDN-of-GitServer:theRepositoryName
```

نکته: در شبکه های بزرگ، معمولن همه ی پرسنل، به همه ی Repo های موجود در Server دسترسی ندارند، پس برای دسترسی به Repo موردنظر، به شکل زیر عمل می کنیم.

مثال: **برای Clone گرفتن از Repo ای به نام Scripts که در پوشه ی Win ، و بر روی Server ای با آدرس** git@git **قرار دارد:**

Git clone git@git:win/scripts

نکته: جهت وارد کردن آدرس در محیط Git Bash، به شکل زیر (مثلن برای پوشه ی ssh.) عمل می کنیم:

/c/Users/USERNAME/.ssh/

: Repo exchange with a Git Server

- در اینجا **git server** را وبسایت **github** درنظر می گیریم؛ هرچند که رویه ی کار برای اتصال به همه ی **Server** ها یکسان است.
- فرض را بر این می گیریم که **SSH Key** که بر روی سیستم ساخته بودیم را در **Server** نیز ثبت کرده ایم.

ایجاد تغییر روی یک Repository موجود در Github:

جهت کار بر روی یک Repo موجود در Github:

- ✓ **ابتدا آدرس SSH آن را پیدا می کنیم.**
- ✓ **Command Line** (Git bash یا Terminal) **را به مسیر local دلخواه برده و به شکل زیر وارد می کنیم:**

```
Git clone git@github.com:payam-avarwand/Test-Git.git
```

ارسال تغییرات local Repository بر روی نسخه ی Remote آن در Github:

- ✓ **ابتدا یک Repo ی خالی در Github، همنام با local repo ای که داریم، ایجاد می کنیم.** (مثلن Test1)
- ✓ **آدرس SSH این Repo را پیدا می کنیم.**
- ✓ **Command Line** (Git bash یا Terminal) **را به مسیر local Repo موردنظر برده و به شکل زیر وارد می کنیم:**

برای Branch اصلی:

```
git push git@github.com:payam-avarwand/Test1.git master
```

و یا

```
git push git@github.com:payam-avarwand/Test1.git main
```

برای هریک Branch دیگر:

```
git push git@github.com:payam-avarwand/Test1.git BranchName
```

می بینیم که:

- **فایل هایی که خود فایل gitignore. هم منتقل می شود، اما فایلها یا Directory هایی که در لیست آن قرار دارند، خیر.**
- **به سادگی در GUI تحت online Git Server ها، می توان Repo ها را ویرایش و commit کرد.**
- **به همین شکل نیز می توان با Git Server های دیگر ارتباط برقرار کرد.**

مثلن برای push کردن یک local Repo به نام Scripts، **به سمت Server ای با آدرس** git@git، و درون پوشه ی Win:

```
Git push git@git:win/scripts master
```

نکته: دستور push، تنها تغییراتی که commit کرده باشیم را منتقل می کند.

می بینیم که جهت push کردن، Repo، لازم است که هربار، آدرس کامل Repo را وارد کنیم! پس می توانیم، یک نام مستعار برای آدرس آن تعیین کنیم، که در push های بعدی، به سادگی آن را وارد کنیم.

نکته: این نام مستعار، از قبل، به شکل پیشفرض، **origin** در نظر گرفته شده است، و ما می توانیم یک نام مستعار اضافه نیز برای آن تعیین کنیم.

به شکل زیر:

```
git remote add myTest git@github.com:payam-avarwand/Test1.git
```

از این به بعد، در دستورهای push، به جای git@github.com:payam-avarwand/Test1.git **می توان از کلمه ی** mytest **استفاده کرد.**

نکته: برای مشاهده ی این نام مستعار، از دستور زیر استفاده می کنیم:

```
git remote -v
```

می بینیم که این نام مستعار، تنها برای دو قابلیت Push و Fetch معتبر است.

❖ جهت به روزرسانی Remote Repo از وجود یک Branch جدید:

فرض کنید از یک remote Repo یک Clone تهیه کرده و بر روی نسخه ی local آن، یک Branch ایجاد و کارهایی را اعمال می کنیم.
حال قصد داریم این local Repo را همراه با تغییرات جدیدی که اعمال کرده ایم، به سمت Remote آن، ارسال کنیم.
اما چون یک Branch جدید ایجاد کرده ایم که در Remote Repo هنوز وجود ندارد، پس به شکل زیر عمل می کنیم:

```
git push -u FQDN-of-GitServer:theRepositoryName NewBranchName
```

نکته: پارامتر -u مخفف upstream و به معنی ارسال رو به بالا می باشد.

معمولاً این دستور را به شکل زیر وارد می کنیم:

```
git push -u origin NewBranchName
```

نکته: وقتی دستور git log را وارد می کنیم، سطر Head برای هر Branch، حداقل باید به شکل زیر باشد:

xxxxxxx (origin/BranchName, BranchName) CommitMessage

درحقیقت همه ی Branch هایی که Name آنها در این پرانتز کنار هم قرار داشته باشند، یعنی در یک سطح و commit یکسان قرار دارند.

اگر سطر Head برای یک Branch به شکل زیر باشد، به این معنی است که این Branch در Remote Server، وجود ندارد:

xxxxxxx (BranchName) CommitMessage

❖ جهت به روزرسانی Remote Repo از حذف یک Branch:

درحقیقت در این حالت، ابتدا آن Branch را در local Repo حذف می کنیم:

```
git branch -d BranchName
```

سپس آن را با دستور زیر، برای remote Repo حذف می کنیم:

```
git push origin -d BranchName
```

❖ جهت به روزرسانی Remote Repo از تغییرنام یک Branch:

وقتی یک Branch را تغییرنام می دهیم:

ابتدا آن را با نام جدید، مثل یک Branch جدید، push کنیم:

```
git push -u origin BranchNewName
```

سپس آن را که با نام قدیمی، همچنان بر روی Server موجود است، حذف می کنیم:

```
git push origin -d BranchOldName
```

: (Integrated development environment) IDE

علاوه بر ابزارهای cmd ، Git Bash ، Tabby ، Atom و Terminal موجود در **Linux**، می توان از بستر (Platform) های مخصوص برنامه نویسی هم جهت کار بر روی **Repository** های **Git** استفاده کرد.

همچنین اگر به آدرس https://git-scm.com/download/gui/windows برویم، می بینیم که خود وبسایت **Git** هم، بسترهای گرافیکی را جهت کار بر روی **Git** معرفی نموده است.

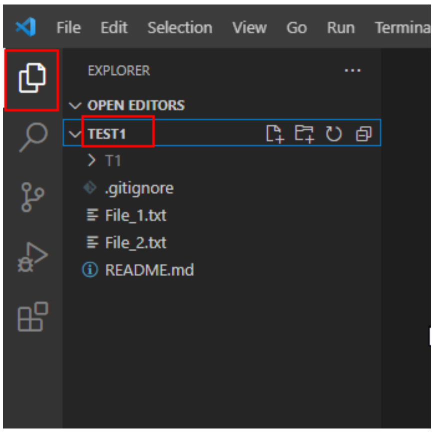
:Visual Studio Code

- در این **IDE**:
  - در **Windows**: می توان هم محیط **PowerShell** و هم محیط **cmd** و هم محیط **Git Bash** و ... را در اختیار داشت.
  - در **Linux**: می توان همه ی نسخه های **Shell** را در اختیار داشت.
- کافی ست در منوی بالا، وارد Terminal > New Terminal شویم.
- برای تغییر **default Terminal**، کافی ست از منوی **View**، گزینه ی **Command Palette** را زده، و در نوار ظاهرشده ی بالا، درج و وارد کنیم `Terminal: Select Default Profile` ، و سپس محیط **CLI** موردنظر را انتخاب کنیم.

برای افزودن یک Remote Repo:

می توان در همان صفحه **Welcome** برنامه (از منوی help در بالا)، گزینه ی clone git repository را انتخاب کرده و آدرس **SSH** را وارد کرد.

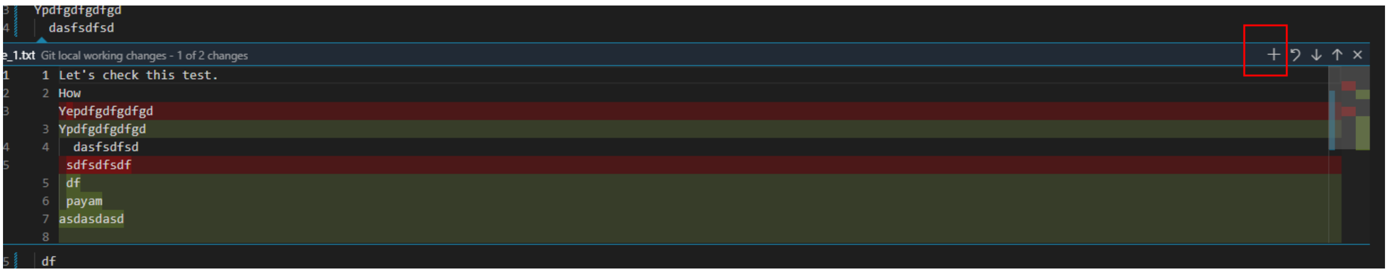
می توان در **Terminal**، خط فرمان را به مسیر موردنظر بُرد، و سپس توسط دستور **Git clone** آن را ذخیره کرد، و سپس توسط File > Open Folder آن را وارد کنیم. به شکل زیر، به محتویات **Repo** دسترسی خواهیم داشت:



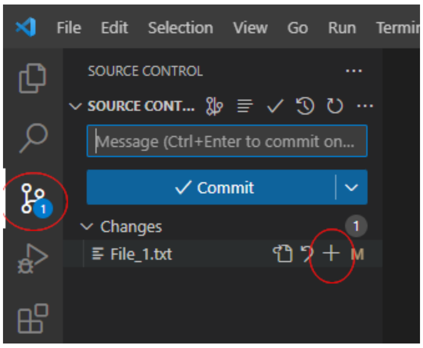
نکات:

- برای اینکه در این نرم افزار بتوان در هر زمان، بر روی بیش از یک Repo کار کرد، باید هریک از Repo ها را درون WorkSpace جاری، Add کرد، به این شکل که در فضای WorkSpace (کادر سمت چپ برنامه) کلیک راست کرده و Add folder to workspace را می زنیم.

به سادگی می توان فایل ها را باز و ویرایش کرد. پس از ویرایش، یک نوار عمودی رنگی، در پشت سطرهای تغییریافته، ظاهر می شود. کافی ست یکبار روی این نوار رنگی کلیک کنیم، تا تغییرات را به دو رنگ قرمز (حذف شده ها) و سبز (اضافه شده ها) نمایش دهد.



اگر بر روی علامت + کلیک کنیم، تغییرات موردنظر را به **Staging area** خواهد فرستاد.
حال برای اینکه تغییرات صورت گرفته روی فایل را ذخیره کنیم، کافی ست کلیدهای **Ctrl+S** را بفشاریم، و یا آن فایل را ببندیم، اما هنگام بستن، در کادر ظاهر شده، با انتخاب **Save**، تغییرات را ذخیره کنیم.

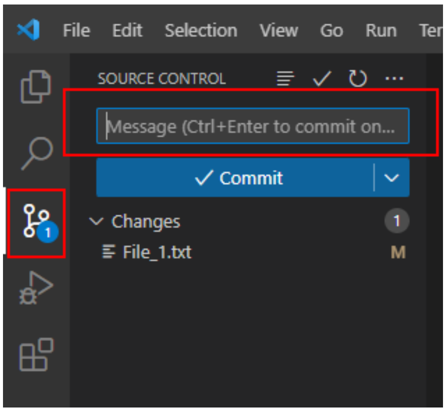


اگر **Staging area** را انجام نداده باشیم و تنها فایل را همراه با تغییرات آن، ذخیره کرده باشیم، به شکل زیر، با کلیک بر روی **+** ، تغییرات اعمال شده را، به بخش **staging area** می فرستیم.

نکته: اگر در زیرمجموعه ی Changes، هیچ آیتمی وجود نداشته باشد، یعنی همه ی تغییرات stage شده اند، پس اگر روی + آن کلیک کنیم، تغییرات را دوباره unstaged می کند!

نکته: البته Save کردن فایلی که تغییر داده ایم نیز به عنوان ارسال به staging محسوب خواهد شد.

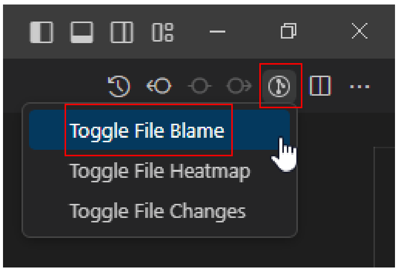
پس از ارسال تغییرات به **Staging Area**، می توان با درج یک **commit message**، عمل **commit** را انجام داد:



Extension های مفید در VSC:

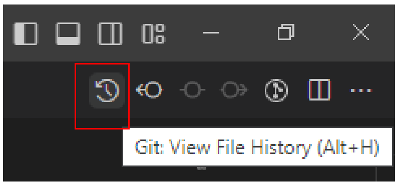
- gitLens

پس از نصب آن، وقتی فایل موجود در یک Repo را در VSC باز می کنیم، کافی ست دکمه ی زیر را بزنیم:



- Git History

پس از نصب آن، وقتی فایل موجود در یک Repo را در VSC باز می کنیم، کافی ست دکمه ی زیر را بزنیم:



: Pycharm

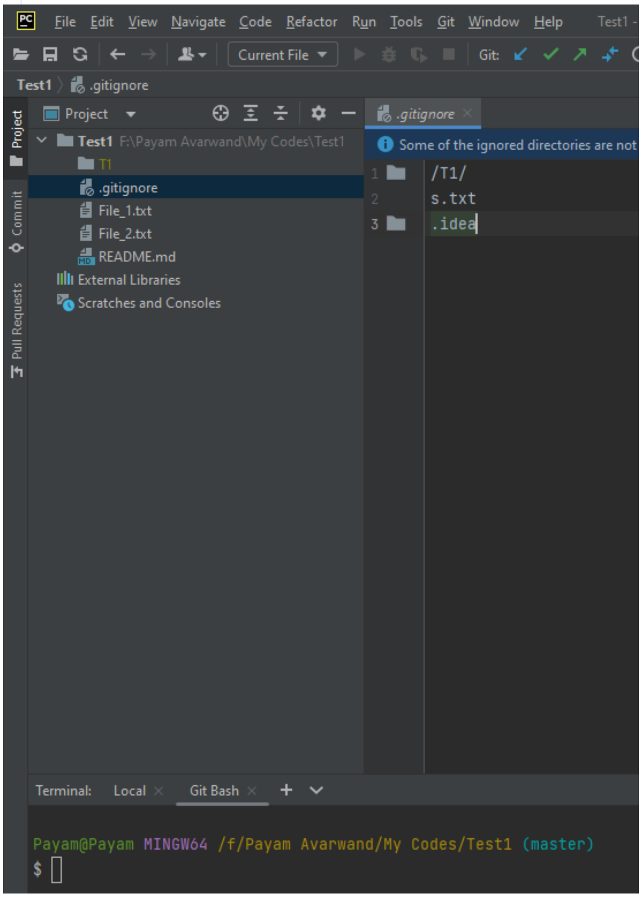
نسخه ی open Source آن، رایگان خواهد بود.
پس از نصب و باز کردن آن، بر روی Get from Version Control کلیک می کنیم.
می بینیم که با حساب GitHub هم می توان در این نرم افزار login کرد.
همچنین می توان به قسمت Repository URL رفته و آدرس SSH مربوط به Repo موردنظر را در نوار URL وارد و دکمه ی Clone را زد.

- **پس از باز کردن Repo در برنامه و باز کردن هریک از File ها، در کادر سمت راست، می توان کلیک راست کرده و با انتخاب گزینه ی Git و از زیرگزینه های آن، Annotate with git blame را زد، تا بتوان تغییرات و Commit ها را بهتر شناسایی و پیگیری کرد.**
- **با کلیک راست بر روی کادر ظاهرشده در سمت چپ، می توان از طریق view، گزینه ی Revision را فعال کرد.**
- **کلیه ی عملیات را Git را می توان در منوی بالای برنامه در زیرگزینه های Git انجام داد.**

❖ **در این IDE هم می توان محیط PowerShell و هم محیط cmd و هم محیط Git Bash را در اختیار داشت.**

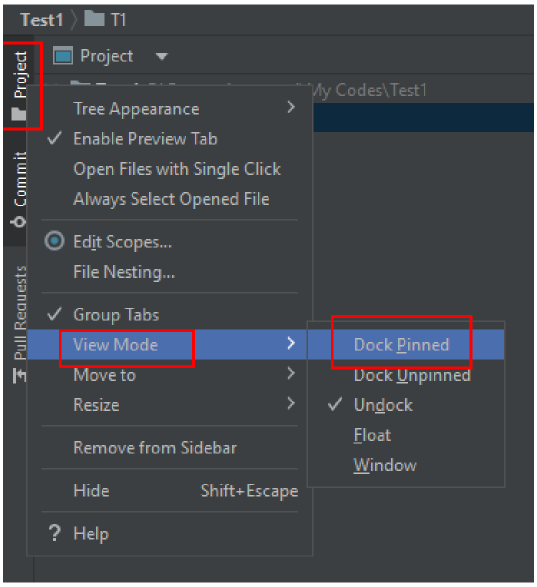
نکته: وقتی Repo را در این برنامه باز می کنیم، می بینیم که پوشه ای به نام idea. بطورخودکار درون Repo ایجاد می شود، که شامل کلیه ی تنظیمات برنامه ی Pycharm برای Repo موردنظر خواهد بود.
اینکه این پوشه را همراه با تمام محتویات دیگر، push کنیم، راه خوبی نیست، چون اگر Repo را در یک سیستم دیگر clone کنیم، این تنظیمات که هیچ کمکی نخواهند کرد، احتمال ایجاد مشکلاتی در شناسایی Repo نیز وجود خواهد داشت.
پس بهترین کار این است که پوشه ی idea. را به لیست gitignore. اضافه کنیم.

جهت افزودن Remote Repo های بیشتر به برنامه، کافی ست از منوی بالا، وارد Git < Clone شویم.
نکته: اگر تابحال، مشخصات مان را بطور global توسط Git config، در سیستم ثبت نکرده باشیم، احتمالاً در اولین commit ای که ثبت می کنیم، تنظیمات git config را سوال خواهد پرسید.
نکته: پس از افزودن اولین Repo، جهت افزودن پروژه های بعدی، لازم است که از منوی بالا، وارد Git شده و Clone را انتخاب و باز کرد، که یا Repo قبلی را می بندد و یا Repo جدید را در یک پنجره ی جدید از Pycharm باز می کند. و برای اینکه در دفعات بعدی، میان Repo ها جابجا شویم، می توان از File < Recent Projects اقدام کرد، و یا به سادگی: File < Open



درحالت ایده آل، محیط برنامه، باید به شکل مقابل به نظر برسد:

جهت اینکه تب های سمت چپ، به شکل عمودی و دایم، قرار بگیرند:



Git Merge :

فرض کنید چند Branch موجود هستند و هریک دارای تغییراتی نسبت به Master Branch بوده و حال قصد داریم تعدادی یا همه ی Branch ها را با master Branch ادغام کنیم.

دو حالت برای merge وجود دارد:

- **Fast forward**: در این حالت Branch ای که می خواهیم با master ادغام شود، یک یا چند commit ثبت شده دارد، درحالی که خود master، همچنان بی تغییر و مثل زمانی است که Branch از آن، انشعاب دریافت کرده است.
  در این حالت، احتمال بروز conflict وجود ندارد.
- **No fast forward**: در این حالت، پس از منشعب شدن Branch از master، خود master دچار تغییراتی شده و یک یا چند commit ثبت کرده است.
  در این حالت، احتمال بروز conflict، پس از درخواست merge، وجود دارد.
  در این حالت، در صورت عدم وجود conflict، پس از واردکردن دستور merge، به شکل خودکار، درخواست درج یک commit message را خواهیم دید.

1. **ابتدا master Branch را انتخاب می کنیم** (وارد آن می شویم).
2. **دستور زیر را وارد می کنیم:**

```
git merge BranchName1 BranchName2
```

درحقیقت آخرین Commit ها، جایگزین commit های قدیمی تر می شوند.
حال که همه ی تغییرات آن Branch ها، با master ادغام شده اند، اما همچنان خود آنها وجود دارند، می توان آن Branch ها را حذف کرد.

نکته: اگر بخواهیم دو یا چند Branch غیر master را با هم merge کنیم، کافی ست وارد یکی از آن Branch ها شده و دستور زیر را وارد کنیم:

```
git merge otherBranchName1 otherBranchName2
```

Git pull :

گاهی، یکی از Branch ها در remote repo، دچار تغییراتی می شود.

پس در local Repo، جهت دریافت آن تغییرات از دستور pull استفاده می کنیم. به این شکل:

1. **ابتدا CL را وارد Branch موردنظر می کنیم.**(مثلن: Branch-1)
2. **سپس دستور زیر را وارد می کنیم.**

```
git pull origin Branch-1
```

پس به سادگی همه ی تغییرات، بر روی محتویات local Branch-1 اعمال خواهند شد.

اما همیشه لازم است که یکبار git status بگیریم، تا از عدم وجود Conflict مطمئن باشیم.

نکته: حال فرض کنید CL را وارد Branch-2 کرده و سپس دستور بالا را وارد کنیم؛ پس تمام تغییرات مربوط به Branch-1 را بر روی Branch-2 اعمال خواهد کرد.

کاربرد دیگر pull:

فرض کنید، یک تغییر را commit و سپس به سمت Server ارسال (Push) می کنید، اما فورن تصمیم می گیرید یک تغییر کوچک را نیز اعمال و آن را بر روی آخرین Commit ای که کرده بودید، به شکل amend، اعمال کنید، در این حالت، می توان به سادگی commit amend را اعمال کرد، اما اگر سپس آن را push کنیم، به احتمال زیاد، خطا می دهد که Server دچار یک سری به روزرسانی هاست که شما از آن عقب تر هستید، به همین جهت لازم است که یکبار دستور pull را اعمال کنید، و سپس از نو push را اعمال کنید.

وقتی conflict رخ می دهد، هریک آن بروزرسانی ها، دریافت و در فایل های موردنظرشان، به شکل یک قسمت جدید، در کنار محتویات قبلی، ذخیره می شوند، و آن فایل یا فایل ها، به شکل untracked در local Repo، قرار خواهند گرفت!

نکته: گاهی ممکن است پس از وارد کردن دستور Git pull، عملیات Pull انجام شود، اما با درج دو خط زیر، در انتهای پاسخ ظاهرشده، اعلام به عدم merge محتوای pull شده کند:

```
* branch          master       -> FETCH_HEAD
fatal: refusing to merge unrelated histories
```

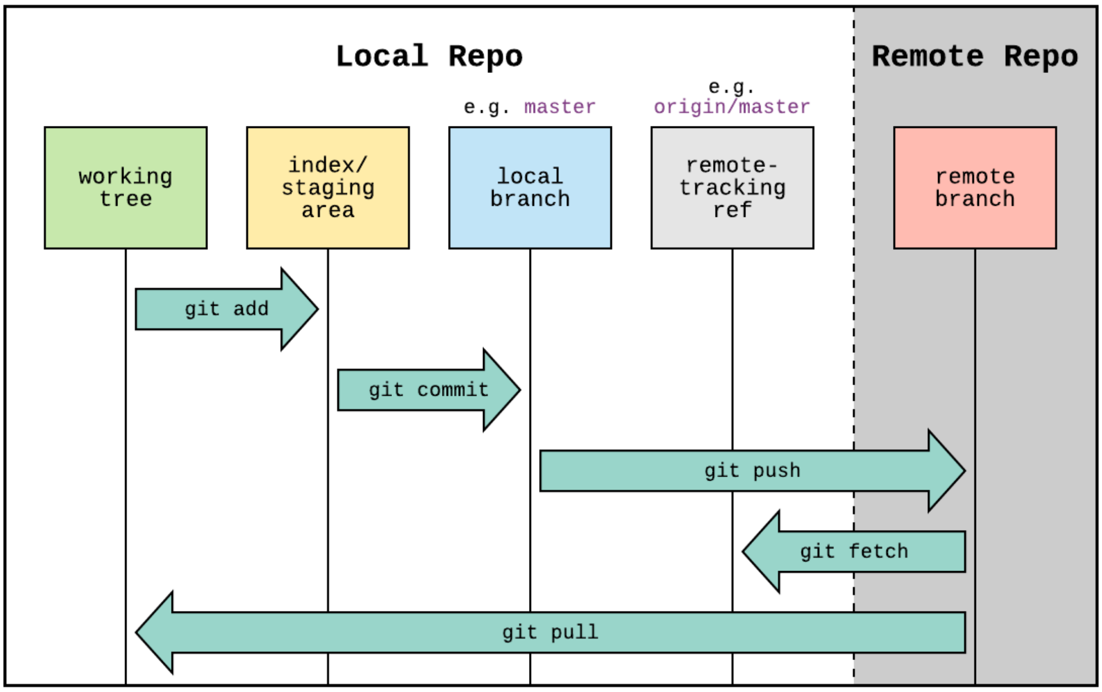
اگر محتوای pull شده، محتوایی که مشابه آن را در local Repo نداشته باشیم، باشد، دستور pull را به شکل زیر اعمال و مشکل را رفع می کنیم:

```
git pull git@github.com:payam-avarwand/TEST.git master --allow-unrelated-histories
```

Git fetch :

می دانیم که دستور **pull** با دریافت تغییرات جدید از **remote Repo**، بطورمستقیم، اقدام به بروزرسانی **local Repo** می کند.
یعنی دستور **git pull** محتوای **remote** را دانلود می کند و بلافاصله به شکل خودکار دستور **git merge** را اجرا می کند؛ و در این حالت احتمال بروز **conflict** وجود خواهد داشت.

اما گاهی لازم است که قبل از به روزرسانی **local Repo**، ابتدا آن تغییراتی که قرار است دریافت و اعمال شوند را، بررسی کنیم.
پس توسط **git fetch** می توان محتوای **remote** را دانلود کرد، اما وضعیت کاری **local** را به روز نکرد.
پس از لحاظ صحت و سلامت، دستور **fetch**، بهتر است.



می بینیم که **git fetch**، به روزرسانی ها را از **remote Repo** دریافت و در یک فضای **local** به نام remote tracking reference ذخیره می کند.
برطبق این قاعده، **git pull** همان **git fetch** است که بعد از عملیات **fetch**، محتویات remote tracking reference را با **local branch** ادغام (**merge**) می کند.

فرض کنید **CL** را در **master Branch** قرار داده ایم.

- دستور زیر:

```
git pull origin master
```

- برابر است با هردو دستور زیر که به ترتیب آنها را وارد کنیم:

```
git fetch
git merge origin/master
```

درحقیقت، زمانی که قصد **merge** کردن **commit** هایی را داریم، بهتر است از **fetch** استفاده کنیم، که به شکل کنترل شده، **merge** کنیم.

Conflict :

وقتی حداقل یک فایل، از دو مسیر مختلف، دچار تغییر شده باشد، و حال سعی در به روزرسانی وضعیت آن فایل بر روی **Server** داشته باشیم، اختلال یا **conflict** پیش می آید.

حالت اول:

فرض کنید شما و یک کاربر دیگر، در **Branch** های متفاوت، بر روی یک **Repo**، کار می کنید.

هردو شما، یک فایل را به شکل های متفاوت، تغییر داده اید؛ پس ابتدا هردوی شما، تغییرات را به سمت **Server**، ارسال یا **push** می کنید.

حال اگر درخواست **merge** این دو **Branch** را بدهیم، اختلال ایجاد خواهد شد؛ طوری که در پاسخ، دریافت می کنیم:

```
CONFLICT (content): Merge conflict in ...
Automatic merge failed; fix conflicts and then commit the result.
```

حال، اگر در هریک از آن **Branch** ها، دستور git status را بزنیم، می بینیم که آن فایل، به حالت untracked (unmerged) تغییر کرده است.

- ✓ اگر بخواهیم علاوه بر این **Conflict**، بطورکل، این فرآیند **merge** را لغو کنیم، دستور زیر را وارد می کنیم، تا همه چیز به حالت قبل بازگردد:

```
git merge --abort
```

- ✓ اما برای رفع **conflict** و تکمیل فرآیند **merge**، کافی ست فایل یا فایل های **unmerged** شده را باز کرده و محتویات هریک را، آنطور که می خواهیم، نگه داشته یا تغییر داده، و بقیه را حذف کرده و فایل را ذخیره کنیم.

  در پایان، آن فایل را **add** و **commit** می کنیم، و می بینیم که **conflict** رفع و **merge** کامل می شود.

حالت دوم:

فرض کنید هم شما و هم یک کاربر دیگر، یک **Repo** را **clone** گرفته و به شکل **local** روی آن کار می کنید.

هر دوی شما یک فایل را ویرایش کرده و تغییراتی متفاوتی را روی آن فایل ایجاد کرده و آن را ذخیره می کنید.

آن کاربر، تغییرات را با موفقیت **push** می کند. پس آن فایل در **Repo** بر روی **Server**، در یک وضعیت به روزتری نسبت به همان فایل در نسخه ی **local clone** شما دارد.

حال اگر شما، **local Repo** را **push** کنید، با خطای [rejected] ! مواجه می شوید؛ که در پایان، پیشنهاد استفاده از دستور git pull را نیز داده است.

1. پس توسط دستور زیر، تغییراتی که اخیرن بر روی آن فایل در **Server** رخ داده اند را درخواست می کنیم:

```
git pull origin master
و یا
git pull origin main
```

2. سپس در پایان، همان فایل را در **local Repo** باز می کنیم و می بینیم که هم تغییرات جدیدی که روی آن اعمال شده بوده است و هم تغییراتی که خودمان اعمال کرده ایم را در بخش های جدا، درج کرده است.
3. به سادگی، هرآنچه که لازم است را نگه داشته و مابقی را حذف کرده و فایل را ذخیره می کنیم.
4. حال لازم است که این تغییر را با ثبت یک **commit**، ثبت کنیم.
5. حال دوباره این **local Repo** را به سمت **Remote Repo** منتقل یا **push** می کنیم و باید بدون **conflict** انجام بشود.

حالت سوم:

فرض کنید یک Branch را Rebase می کنیم، پس تداخلی مشابه با دو حالت قبل رخ می دهد که به سادگی قابل رفع خواهد بود؛ درغیراینصورت، توسط دستور زیر، آن را لغو می کنیم:

```
git rebase --abort
```

نکته: یکی از روش های جلوگیری از Conflict در مورد ارتباط با Online Git Server ها، این است که کاربرها پس از کار بر روی یک Branch از یک Repo، می توانند merge request به سمت آن Repo ارسال کنند، تا صاحب آن Repo، آن را بررسی و تایید یا رد کنند.
البته در Github، به این درخواست، Pull request گفته می شود؛ که صاحب Repo اصلی، درخواست هایی که برای هر Repo، به او ارسال شده اند را در تب pull requests مشاهده می کند.

نوشته ی
پیام اوَروند
زمستان 2023 - 1401

- Payam_Avar@yahoo.com
- @payam.avarwand